2020-12

BOUNDARY SEARCH AND TRAVERSE: DETERMINING THE THRESHOLD BOUNDARY FOR INFRASTRUCTURE NETWORK RESILIENCE PROBLEMS

Flinn, Lewis C.

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

BOUNDARY SEARCH AND TRAVERSE: DETERMINING THE THRESHOLD BOUNDARY FOR INFRASTRUCTURE NETWORK RESILIENCE PROBLEMS

by

Lewis C. Flinn

December 2020

Co-Advisors:	David L. Alderson Jr.
	Ralucca Gera
Second Reader:	W. Matthew Carlyle

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2020	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
BOUNDARY SEARCH AND TRAVERSE: DETERMINING THE THRESHOLD BOUNDARY FOR INFRASTRUCTURE NETWORK RESILIENCE PROBLEMS

5. FUNDING NUMBERS

6. AUTHOR(S) Lewis C. Flinn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Traditional network interdiction problems focus on finding the target package that yields the worst-case performance for network operations given a level of attack. However, in many cases it may be desirable to disrupt a network such that its performance falls below a given performance threshold while retaining other properties, such as the ability to quickly restore performance. This is important in traditional joint operations, where the military may conduct shaping operations by interdicting a network to enable tactical success, then conduct stabilization operations that require restoration of the network. This broader view of network shaping is important for exposing operationally relevant states that are not normally identified by the traditional interdiction and restoration methods of network flow problems. However, unlike attacker-defender problems that can be solved efficiently using appropriate algorithms, identifying the complete boundary that separates operational from non-operational states requires some form of enumeration, which may take too long to be useful in joint military operations. This thesis considers the use of graph theoretic measures, such as edge betweenness centrality and modularity, to develop heuristic methods for enumerating the boundary between operational and non-operational states.

14. SUBJECT TERMS network, boundary, interdiction, shaping	**15. NUMBER OF PAGES** 87
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

BOUNDARY SEARCH AND TRAVERSE: DETERMINING THE THRESHOLD BOUNDARY FOR INFRASTRUCTURE NETWORK RESILIENCE PROBLEMS

Lewis C. Flinn
Major, United States Marine Corps
BA, Bowdoin College, 2010

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

and

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
December 2020

Approved by: David L. Alderson Jr.
Co-Advisor

Ralucca Gera
Co-Advisor

W. Matthew Carlyle
Second Reader

W. Matthew Carlyle
Chair, Department of Operations Research

Wei Kang
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Traditional network interdiction problems focus on finding the target package that yields the worst-case performance for network operations given a level of attack. However, in many cases it may be desirable to disrupt a network such that its performance falls below a given performance threshold while retaining other properties, such as the ability to quickly restore performance. This is important in traditional joint operations, where the military may conduct shaping operations by interdicting a network to enable tactical success, then conduct stabilization operations that require restoration of the network. This broader view of network shaping is important for exposing operationally relevant states that are not normally identified by the traditional interdiction and restoration methods of network flow problems. However, unlike attacker-defender problems that can be solved efficiently using appropriate algorithms, identifying the complete boundary that separates operational from non-operational states requires some form of enumeration, which may take too long to be useful in joint military operations. This thesis considers the use of graph theoretic measures, such as edge betweenness centrality and modularity, to develop heuristic methods for enumerating the boundary between operational and non-operational states.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AD	Attacker-Defender
BST	Boundary Search and Traverse Algorithm
ABC	Arc Betweenness Centrality
AEC	Arc Eigenvector Centrality
DAD	Defender-Attacker-Defender
DOD	Department of Defense
FIFO	First-In First-Out
GN	Girvan-Newman Community Detection Algorithm
NPS	Naval Postgraduate School
USN	U.S. Navy
USG	United States government

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Analyzing critical infrastructure networks is important from both an offensive and defensive perspective. Governments conduct defensive measures to ensure a network provides a minimum level of performance, and militaries conduct offensive measures to reduce an adversary's network performance. In the recent wars in Iraq and Afghanistan, the U.S. military destroyed critical infrastructure only to have to rebuild it later. Recent work reveals this fundamental issue and refers to the two-part problem as *network shaping*, which identifies beneficial network system operating conditions that are not normally identified by traditional optimization methods. The key idea is determining what future operating conditions, or *states*, are possible and which of those are most advantageous for a particular military action.

Unfortunately, this type of analysis relies upon evaluating the network's performance in the various operating conditions, which grow exponentially with the number of connections between components. This motivates us to answer the following question: *How can we find these important operating states in a network system without resorting to exhaustive enumeration?*

This thesis examines different techniques to enumerate a *metagraph* of states, representing all possible operating conditions, to reveal useful states that are not typically identified by traditional methods. We look to well-known measures in graph theory, such as edge betweenness centrality and modularity, for potential insights into key network structure, and also consider a variety of techniques for reducing computational requirements.

Our solution technique follows from two key recognitions. First, we typically don't need to know the operating performance of every possible system state, but instead are often only concerned with whether the performance meets a given performance *threshold*. We refer to performance above the threshold as *mission success* and performance below the threshold as *mission failure*. Then, the real problem is to separate the success states from the failed states, and we do this by finding the *boundary* between them.

The second key recognition comes from an assumption that system performance decreases monotonically with each additional damaged component. Thus, if any system configura-

tion can be identified as failed, then all configurations with additional damage will also correspond to mission failure. This allows us to reduce the number of scenarios we have to evaluate.

We propose a new algorithm called *Boundary Search and Traverse (BST)* that navigates through the space of possible network states and quickly partitions this space by following the boundary for a particular network performance threshold. Although the BST algorithm has memory requirements above what is required for exhaustive enumeration, it requires considerably fewer scenario evaluations.

We test and validate our technique with a number of small, but representative, networks, and we then apply the BST algorithm to a historical railroad network of realistic size. We explore the runtime tradeoffs associated of different computational implementations.

Overall, this thesis brings the concept of network shaping closer to operational employment. However, there is still an opportunity for additional contributions that identify and incorporate more efficient data structures. The use of machine learning and other heuristic techniques also hold promise for finding solutions in relatively short time.

Acknowledgments

I would like to sincerely thank the members of my advisory team for their support throughout the thesis process and preceding coursework. It has been a pleasure and a great learning experience to work with Dr. Alderson, Dr. Gera, and Dr. Carlyle at the Naval Postgraduate School.

I would also like to thank the Senior Marine Office, particularly LtCol Forbell for his support and mentorship both inside and outside of the classroom.

To my fellow students, I learned a lot from you all, particularly Dave Barnhill and Kevin Lutz.

To Kali and my brother, John, thank you for proofreading my various papers and briefs, and for all of the memorable adventures in Monterey and the surrounding national parks.

Finally, I would like to thank my parents for their continued support throughout my career. I could not have asked for a kinder upbringing, and I love you both.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Networks are present throughout many aspects of our daily lives. We often hear of computer networks and transportation networks, but we can also represent many types of critical infrastructure systems as networks. To describe these systems, we adopt the terminology of Alderson et al. (2014, p. 182): "An infrastructure [network is] a system of interconnected components that work together to provide a particular function. Examples of function include traffic conveyance, electric power transmission, fuel delivery, manufacturing, supply chains, and communication." Analyzing these critical infrastructure networks is important from both an offensive and defensive perspective. Governments conduct defensive measures to ensure a network provides a minimum level of performance after sustaining a random or targeted disruption; defensive measures are restorative. Offensive measures, often useful during military operations, are intended to reduce an adversary's network performance and resilience, making the network ineffective and fragile; offensive measures are damaging.

1.1 Interdiction in Military Operations

Traditional joint military operations are composed of six phases: Shape, deter, seize initiative, dominate, stabilize, and enable civil authority (Joint Chiefs of Staff 2018):

- **Phase 0, Shape:** "Help set conditions for successful theater operations"(Joint Chiefs of Staff 2018).
- **Phase I, Deter:** "Prevent an adversary's undesirable actions, because the adversary perceives an unacceptable risk or cost of acting"(Joint Chiefs of Staff 2018).
- **Phase II, Seize Initiative:** Get the upper hand through defensive and offensive actions.
- **Phase III, Dominate:** Overmatch the enemy's capabilities to "break the enemy's will to resist"(Joint Chiefs of Staff 2018).
- **Phase IV, Stabilize:** "Help reestablish a safe and secure environment and provide essential government services, emergency infrastructure reconstruction, and humanitarian relief"(Joint Chiefs of Staff 2018).
- **Phase V, Enable Civil Authority:** "Support legitimate civil governance...to help the civil authority regain its ability to govern and administer the services and other needs

of the population"(Joint Chiefs of Staff 2018).

As stated in *Joint Publication 5-0: Joint Planning* (Joint Chiefs of Staff 2017), "shaping the [Operational Environment] is changing the current conditions within the [Operational Environment] to conditions more favorable to U.S. interests. It can entail both combat and noncombat operations and activities to establish conditions that support future U.S. activities or operations." Additionally, "shaping activities may begin during plan development to help set conditions for successful execution [and] they may continue after the operation ends" (Joint Chiefs of Staff 2018). Therefore, we can view shaping as both the attack on a network to enable tactical success, and the restoration of the network to conduct stabilization operations and enable civil authority; the military begins the operation as the network attacker and ends as the network defender. In the recent wars in Iraq and Afghanistan, the U.S. military has destroyed critical infrastructure only then to have to rebuild it. As stated in Barrow (2019), "this process of 'break it bad' then 'fix it fast' is inherently inefficient (Hart et al. 2014)."

Though this concept may seem novel to military operations, the underlying ideas have been around for quite some time. Oltman et al. (1996, p. 150) discuss "the need to preserve infrastructure. . . yet still delay, disrupt, divert, or destroy" targets during conventional and non-conventional missions. Although Oltman et al. (1996) are primarily advocating for materiel advancements to achieve improved precision and accuracy for future munitions, they recognize that a key factor in achieving the desired effects of precision and lethality is information dominance. They envision a virtual version of Boyd's famous OODA Loop (Figure 1.1) that employs "vast computational power, with algorithms capable of simulating the chaotic nature of events as they unfold" since war is chaotic and requires nonlinear modeling (Oltman et al. 1996, p. 151).

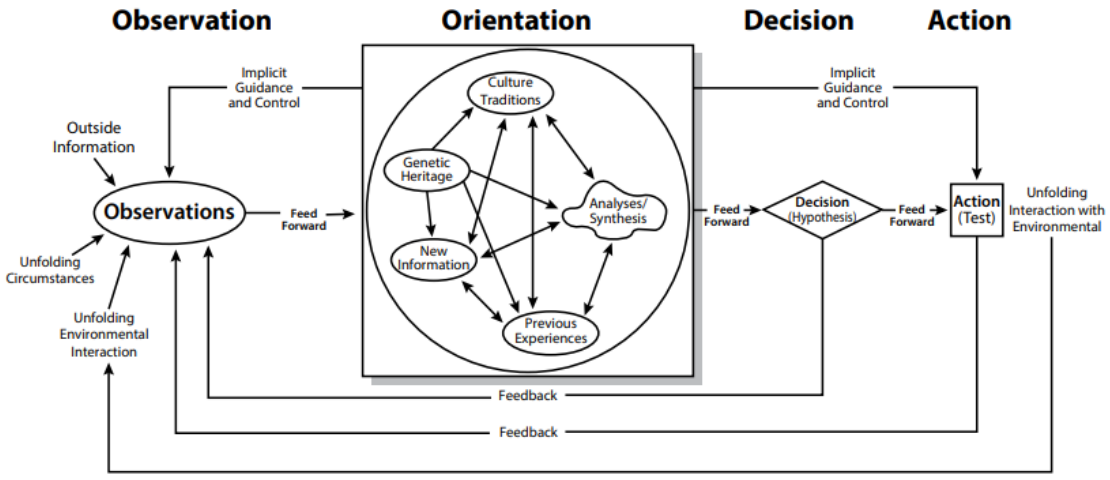


Figure 1.1. Boyd's Observe-Orient-Decide-Act Decision Cycle. Boyd's OODA Loop is a conceptualization of how to mentally maneuver vis á vis an enemy. Though often reduced to the importance of relative speed in recognition and decision, Boyd's model contains multiple feedback loops that represent the importance of timely and proper perception of reality and how implicit decision-making can be positive (if perception is correct) or negative (if perception is incorrect). Source: Boyd (2018).

Ultimately, "the ability to model the battle space and explore options before actually executing them will allow analysis to determine best effects for least cost. . . such a system. . . would allow commanders to plan interdiction sorties against centers of gravity for the upcoming enemy operation" (Oltman et al. 1996, p. 160). The key idea is determining what future operating conditions, or *states*, are possible and which of those are most advantageous for a particular military action.

1.2 Network Shaping for Military Operations

In a recent thesis, Barrow (2019) introduces the *network shaping* problem as follows: "Are there system states not normally identified via traditional [interdiction/restoration] methods that are operationally relevant and inform new approaches to joint operations?" (Barrow 2019, p. xv). Traditional interdiction and restoration methods focus on the most harmful interdiction and best restoration, but these individual optima may not be the best paired actions for military operations (Barrow 2019, p. 43).

Barrow (2019) analyzes the performance of an example network under various operating conditions to show that there are useful circumstances in which the network performs below a critical performance threshold, but the network can also be expeditiously fixed to bring performance back above the threshold. An operating condition, or *network state*, like this could be useful in military operations when considering Phases IV and V, Stabilize and Enable Civil Authority. Barrow (2019) also discusses *fragile states* where "any additional interdiction results in a transition across the performance boundary . . . [and] where system performance meets mission requirements, but is fragile to any additional loss in the original flow network" (Barrow 2019, p. 30). He also explains the importance of states with opposite characteristics: "Restoration of any single broken component in the flow network returns the system to a state where performance meets mission requirements" (p. 30).

Unfortunately, analysis like that in Barrow (2019) relies upon evaluating the network's performance in the various operating conditions, which grow exponentially with the number of connections between components. Alderson and Carlyle (2017) propose a "smart" enumeration technique that reduces the amount of work required, but their results are only preliminary and more work is required to make these concepts practical for use in military operations.

1.3 Thesis Objectives

The goal of this thesis is to find these important operating states in a network system without resorting to complete enumeration. Specifically, we look to well-known results in graph theory for potential insights into key network structure, and also consider a variety of techniques for reducing computational requirements.

In Chapter 2, we discuss related literature and examine potential methods to reduce computational requirements for determining network performance. In Chapter 3, we build on past work and present a novel algorithm for finding these states. In Chapter 4 we compare the performance of this algorithm against existing enumeration schemes for a number of test networks, and then apply them to a realistic transportation system. In Chapter 5, we summarize our results and present opportunities for future work.

Our ultimate goal is to enhance the concept of network shaping and make it more accessible for planners of military operations.

CHAPTER 2:
Background

We review three classic types of network flow problems and their extension to network interdiction and defense. We then introduce additional concepts and a lexicon to support our analysis of network restoration and shaping activities, and the computational issues associated with these problems. Finally, we discuss techniques and measures to address the computational complexity of problems requiring complete enumeration.

2.1 Network Flow Problems

Network flow problems are mathematical constructs designed to answer questions about how to move some commodity from one point to another. The networks are composed of nodes, which are the various supply, demand, and transfer locations, and arcs, which are the connections between nodes. There are three classic types of network flow problems: *minimum cost flow, shortest path, and maximum flow* (Ahuja et al. 1993).

Minimum Cost Flow Problem. Ahuja et al. (1993) describe the *minimum cost flow problem* as finding the lowest cost for shipping a commodity through a network while satisfying the demands at certain nodes and using available supply at other nodes. Specifically, the model is formulated below using NPS format (Brown and Dell 2007):

Index Use [cardinality]

$i \in N = \{1, 2, ..., n\}$ nodes in network $[|N| = n]$

$(i, j) \in A = \{(1, 2), ..., (i, j), ...\}$ arcs in network $[|A| = m]$

Given Data

c_{ij} cost per unit flow over arc (i, j)

l_{ij} minimum allowable flow over arc (i, j)

u_{ij} maximum allowable flow over arc (i, j)

$b(i)$ supply of unit at node i; if $b(i) < 0$ then there is demand at node i

Decision Variables

x_{ij} amount of flow to send over arc (i, j)

Formulation

$$Minimize \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.1}$$

subject to

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(i,j)\in A} x_{ji} = b(i), \forall i \in N, \tag{2.2}$$

$$l_{ij} \leq x_{ij} \leq u_{ij}, \forall (i, j) \in A, \tag{2.3}$$

where $\sum_i^n b(i) = 0$ (i.e., supply = demand). Equation 2.1 is the objective function and represents the cost of flow across all arcs. Equation 2.2, commonly referred to as the *conservation of flow*, ensures that for each node i, the sum of its outflows, x_{ij}, minus the sum of its inflows, x_{ji}, is equal to its supply, $b(i)$. Equation 2.3, the *arc capacity constraint*, ensures that the flow across any arc remains within the lower and upper bounds, l_{ij} and u_{ij}, respectively.

Equation 2.1 can also be represented in the more compact, vector notation (Alderson et al. 2015) for what is often called the *operator problem*:

$$\min_{\mathbf{y}\in Y(\hat{x})} f(\hat{x}, \mathbf{y}) \tag{2.4}$$

"where $\hat{x}$ is a vector that collectively represents whether each of the components (the links in our example) in the system is working or broken (also called the operating state), the set $Y(\hat{x})$ represents the feasible actions of the operator (here, allowable flows) for given state $\hat{x}$

of the system, and $f(\hat{x}, \mathbf{y})$ is a function that measures the performance (here, the cost) that results from the choice of activities $\mathbf{y}$" (Alderson et al. 2015).

Throughout this thesis, we use the term *up* to refer to arcs that are working (operational), and we use the term *down* to refer to arcs that are broken (interdicted, attacked). Thus, the operating state for a flow network is defined by the set of "up arcs" and the set of "down arcs."

Shortest Path Problem. The objective of the *shortest path problem* is to find the minimum length (or cost) from a particular origin (or source), *node s*, to a specified destination (or sink), *node t*. We set the supply of *node s* to one and the supply of *node t* to negative one to represent demand; for all other nodes, the supply is set to zero: $b(s) = 1$, $b(t) = -1$, and $b(i) = 0, \forall i \neq s, t$. We then solve for the minimum cost flow solution (Ahuja et al. 1993).

Maximum Flow Problem. The objective of the *maximum flow problem* is to find the largest possible flow between a particular source, *node s*, and a specified sink, *node t*, without exceeding the capacities of any arcs. The arc capacity, u_{ij}, is the maximum flow rate from *node i* to *node j*. We set all node supplies and all arc costs to zero, $b(i) = 0, \forall i \in N$ and $c_{ij} = 0, \forall (i, j) \in A$, then introduce a new arc from *node t* to *node s* with unlimited capacity and an arc cost of negative one, $u_{ij} = \infty$ and $c_{ij} = -1$. We then solve the minimum cost flow solution which provides the maximum flow along arc (t, s) and is equal to the maximum flow through the network from *node s* to *node t* (Ahuja et al. 1993).

2.1.1 Network Interdiction: "Attacker" Models

In network interdiction, an *attacker* wishes to target arcs in such a way as to deteriorate the operator's optimal solution. For a shortest path problem, the attacker seeks to lengthen the operator's shortest path; for a maximum flow problem, the attacker seeks to minimize the maximum flow; and for a minimum cost problem, the attacker seeks to maximize the minimum operating cost. Choosing which arcs to target is equivalent to choosing the set of down arcs for the operating state.

When targeting an arc in one of these flow problems, the effect may be a limit on the flow capacity for the particular arc (possibly to zero), or increasing the cost for using the arc (possibly up to a pseudo-infinite number). Ultimately, re-solving the original problem

results in a re-routing of flow. We represent this bi-level problem in the same vector notation (Alderson et al. 2015):

$$\max_{\mathbf{x}\in X} \min_{\mathbf{y}\in Y(\mathbf{x})} f(\mathbf{x}, \mathbf{y}). \tag{2.5}$$

In this problem, the attacker chooses $\mathbf{x}$, which the operator then incorporates in the *operator sub-problem*. Owing to its sequential nature, the problem in (2.5) is sometimes called an Attacker-Defender (AD) model.

Although we refer to these as *attacker models*, the notion of an "attacker" is conceptual. There might not be an actual attacker, but random events (e.g., natural disaster, component failure) may still occur and these models help find the worst-case disruption, even after re-routing.

2.1.2 Network Defense: "Defender" Models

Alderson et al. (2015) describe the *Defender Model* as "mitigating the worst-case operating cost that can result from the simultaneous loss of components" by making improvements with limited resources. We incorporate these actions into our formulation by reducing the effect of an attack on an arc or by increasing the cost of an attack on an arc. The goal is to encourage the attacker to target a different arc or set of arcs. Brown et al. (2006) explain it as the defender wishing to identify a defense plan that minimizes the worst damage the attacker can inflict. Brown et al. (2006) note that these tri-level, defender-attacker-defender models are difficult to solve and discuss certain defense problems that can use bi-level, defender-attacker models. Here we represent the tri-level model for a minimum cost flow problem in the same manner as Alderson et al. (2015):

$$\min_{\mathbf{w}\in W} \max_{\mathbf{x}\in X} \min_{\mathbf{y}\in Y(\mathbf{w},\mathbf{x})} f(\mathbf{w}, \mathbf{x}, \mathbf{y}), \tag{2.6}$$

where $\mathbf{w}$ is the decision vector representing which arcs the defender chooses to defend. Similarly, problem in (2.6) is often called a Defender-Attacker-Defender (DAD) model.

2.1.3 Interdiction Models Provide Extreme Points

Interdiction models, such as AD and DAD formulations, find the extreme worst-case or best-case scenarios. These are of vital importance for assessing vulnerabilities and mitigating them. However, during shaping operations there may be other states of great interest beyond these extremes.

2.2 A Metagraph of States

Most literature for network flow problems focus on how to directly solve the problems or equivalent formulations. However, when considering how to improve or degrade a network over successive actions, it may be more useful to consider how to get from the current state to the desired end-state. Alderson and Carlyle (2017) introduce the concept of viewing the state space for the possible configurations under which the flow network can operate as a metagraph.

2.2.1 Basic Concepts

The *metagraph* represents the state space for the possible configurations under which the flow network can operate. The vertices of the metagraph correspond to the individual system states while transitions in this graph correspond to breaking and repairing arcs in the underlying flow network. In the simplest case where each arc in the flow network is either *up* (operative) or *down* (inoperative), there are 2^m possible states for a flow network with m arcs. Moreover, we assume that only a single arc in the network flow problem can "go up" or "go down" at a time. Therefore, each state has exactly m incident edges or potential transitions to other states.

Throughout this thesis we use *vertices* and *edges* as terminology for the metagraph, while using *nodes* and *arcs* for the underlying flow network.

Let X denote the set of states (indexed $x \in X$) corresponding to the vertices in the metagraph. Let $p(x)$ denote the performance of the network when in state x (as typically evaluated by solving a network flow problem). Given a *performance threshold*, τ, we can partition the states into those that correspond to *mission success* (i.e., $p(x) \geq \tau$) and those that correspond to *mission failure* (i.e., $p(x) < \tau$). More formally, we define the partition of $X = \{\overline{X}, \underline{X}\}$

where $\forall \overline{x} \in \overline{X}$, $p(\overline{x}) \geq \tau$, and $\forall \underline{x} \in \underline{X}$, $p(\underline{x}) < \tau$. By construction, we have $\overline{X} \cup \underline{X} = X$ and $\overline{X} \cap \underline{X} = \emptyset$.

The *cut set* of the partition is the subset of edges $B \subseteq E$ such that $(\overline{x}, \underline{x}) \in B$ implies that $\overline{x} \in \overline{X}$ and $\underline{x} \in \underline{X}$. To avoid confusion with any potential cut set in the flow network, we also refer to the cut set in the metagraph as the *boundary*. That is, the boundary is the partition that separates success states from failure states for a particular performance threshold (Barrow 2019, p. 19). A primary contribution of Barrow (2019) is to characterize states in terms of their distance to the boundary, both in terms of their *performance distance* (i.e., $|p(x) - \tau|$) and their *hop distance* (i.e., minimum number of transitions).

Unlike attacker-defender (AD) problems that can be solved efficiently using appropriate algorithms (e.g., Benders decomposition), identifying the complete boundary currently requires some form of enumeration. Alderson and Carlyle (2017) propose a "smart" enumeration that reduces the amount of work required, but their results are only preliminary.

2.2.2 Network Restoration

Schulze (2014) examines the network restoration problem in which "the network operator is responsible [for finding] a way of reconstructing the system as fast as possible and while using the least amount of resources." Using the same idea of a metagraph of states, network restoration becomes equivalent to finding an attractive path in the metagraph from a current (broken) state to a future (restored) state. He uses two techniques to solve the problem: The first technique is a mixed integer linear program with repairs represented by binary decision variables; the second technique solves the shortest path problem on the metagraph of states, using Dijkstra's algorithm and the A-star algorithm.

2.2.3 Repair or Replace

Brendecke (2016) examines an optimal repair and replacement policy for a multi-component system by formulating a Markov decision problem. His work relates to that in Alderson et al. (2015) by "allowing the links to degrade and fail, and studies when to repair and replace these links to improve system resilience" (Brendecke 2016). The random failure and restoration of individual network components can be interpreted as stochastic drift in the metagraph of states. Brendecke (2016) models network arcs as being in one of three states:

new, old, and failed. As such, he enhances the operator model to have ternary decision variables to represent operator actions: *none, repair, and replace*. He solves the Markov decision problem by formulating an equivalent linear program of steady state probabilities. This technique is effective for small problems, but does not scale well to larger systems.

2.3 Network Shaping

Clark (2017) examines resilience of an *s-t* shortest path network by solving the Threshold Shortest Path Interdiction Problem, which is "how many attacks of a specified cardinality cause the *s-t* shortest path to exceed [a given network performance] threshold." Clark (2017) explains that enumeration is the primary evaluation method for determining network resilience and that enumeration is exponential in nature.

Barrow (2019) examines the attacker and defender models from the standpoint of the military that is responsible for dominating, then restoring the network. Therefore, he seeks to determine if there are system states that are operationally relevant but not normally identified by traditional interdiction and restoration methods (Barrow 2019). Like Clark (2017), Barrow (2019) states that his methods require complete enumeration of a network.

In principle, we could enumerate everything and provide a parametric analysis of the network or identify fragile and resilient states, as done in Clark (2017) and Barrow (2019), respectively. In practice, complete enumeration is not feasible since the problem grows exponentially with the number of arcs in the network: A minimum-cost flow problem for a network with 18 edges must be solved $2^{18} = 262,144$ times (Barrow 2019). In order to operationalize the ideas of network shaping or parametric analysis of resilience, we must find a substitute for complete enumeration.

2.3.1 Computational Issues

Computational complexity is a pervasive challenge in network problems, and many fields have come up with intelligent ways to balance accuracy with computational requirements. Here, we discuss approximation techniques, sub-tree marking, and heuristic measures.

Approximations

For a network of n nodes and m arcs, the traditional maximum flow problem can be solved in $O(nm^2)$, using the Edmonds-Karp algorithm, and in $O(n^2m)$, using the successive shortest path algorithm (Ahuja et al. 1993, p. 240-241). However, if the original flow problem can be solved in less time, then the overall cost of having to enumerate all solutions can be reduced.

Daitch and Spielman (2008) implement solvers for Laplacian linear systems and interior-point algorithms to approximate maximum flow and minimum cost solutions for lossy networks in $O(m^{\frac{3}{2}}log(\frac{1}{\epsilon}))$ time. Christiano et al. (2010) build upon Daitch and Spielman's work by developing a fast algorithm for computing approximate maximum *s-t* flows and minimum cuts by using a multiplicative weights update method and solving electrical flow problems for capacitated, undirected graphs.

Sub-Tree Marking

Current methods to define network performance boundaries rely upon enumeration. Initial analysis focused on exhaustive enumeration to expose key concepts of fragile and resilient states, and distance to the threshold, Threshold Hamming Distance (Barrow 2019, p. xiv). Alderson and Carlyle (2017) improved computational efficiency by assuming that network "performance decreases monotonically with each additional failed [or down] component" and proposing a marking scheme for nodes that are guaranteed to be above the performance threshold (denoted "green") and nodes that are guaranteed to be below the performance threshold (denoted "red"), and they develop conditions under which it becomes possible to mark entire sub-trees in the metagraph. They subsequently adapted their algorithm to use multiprocessing and employ multiple cores which further reduces computation time. Although their preliminary results show significant reduction in computation time, their methods still rely upon some form of enumeration. In order for the concept of network shaping to become operational, we need something that is faster computationally.

Heuristic Measures

Heuristic measures—if they can be validated against complete enumeration and the "smart" enumeration schemes proposed by Alderson and Carlyle (2017)—may provide sufficiently

accurate and timely results to operationalize network shaping. These heuristic measures may come from network interdiction, graph theory, or machine learning areas of research.

2.4 Graph Theoretic Measures and Algorithms

The origins of graph theory hearken back to the famous Seven Bridges of Königsberg problem proposed by Euler in the mid-1700s (see Alderson et al. 2011, for a brief history). In 1959, Paul Erdös and Alfréd Rényi introduced a method for randomly generating graphs to produce a probability distribution for analyzing a real-world based graph (Erdös and Réyni 1959). These models were initially used to examine social interactions, but have recently regained traction in the last several decades. Bollobás (1998) reinvigorated interest in graph theory, and the modern study of Network Science has emerged as a popular field of study for investigating the interactions of complex systems (e.g., see p. 7 Newman 2018, for an introduction). Common measures used to analyze graphs focus how many vertices and edges there are, how far away the vertices are from each other, how central a particular vertex or edge is, how vertices can be organized into communities, and how "connected" the graph is.

2.4.1 Network Characteristics

To characterize networks, we use seven basic terms from graph theory as defined in Chartrand and Zhang (2013). The *order* of a network is total number of nodes, while the *size* of a network is the total number of arcs. To understand how far nodes are from each other, we use *eccentricity*, which is the largest distance between the particular node and any other node in the network. The network *diameter* is the largest eccentricity of any node in the network, while the *radius* of the network is the is the smallest eccentricity of any node in the network. For a given network G, the *connectivity* of the network, $\kappa(G)$, is the smallest number of nodes for which their removal would *disconnect* the network, or split into isolated sub-networks. Similarly, the *arc connectivity* of the network, $\lambda(G)$, is the smallest number of arcs for which their removal would disconnect the network.

2.4.2 Centralities

Centralities are a quantitative way to describe the "importance" of a vertex in a graph and can be broken into two subgroups: adjacency-based and distance-based. Adjacency-

based centralities focus on the number and importance of adjacent vertices. Distance-based centralities focus on the path length from one vertex to another. Here we describe a few common centralities (Newman 2018, serves as a general reference).

- **Degree Centrality** uses the degree of each vertex (i.e., to how many other vertices is a particular vertex directly connected?).
- **Eigenvector Centrality** is a recursive measure that assigns weights to connections based on the importance of the neighbors (i.e., how many other *important* vertices is a particular vertex directly connected to?).
- **Arc Eigenvector Centrality** is a recursive measure that assigns weights to arcs based on the importance of the neighboring arcs (i.e., how many other *important* arcs is a particular arc adjacent to?).
- **Katz Centrality** is similar to Eigenvector Centrality for directed graphs (i.e., how many connections to other *important* vertices with high out-degree?).
- **Closeness Centrality** focuses on the efficiency of a vertex to reach all other vertices.
- **Betweenness Centrality** measures how many shortest paths among all vertices pass through a given vertex.
- **Edge Betweenness Centrality** measures how many shortest paths among all vertices pass along a given edge.

When we consider the use of these measures in network shaping and analysis of flow networks, our intuition points us towards the efficiency measures of closeness and betweenness. Since we are also trying to determine which arcs to "attack" or "defend," we would like an edge-based, graph theoretic measure. *Edge betweenness centrality* is "the number of shortest paths between pairs of vertices that run along" a particular edge (Girvan and Newman 2002, p. 7822). This measure is meaningful when calculated on the flow network and not on the metagraph, since it can be shown that all edges and vertices are equal for any measure on the metagraph. The intuition is that important arcs are "highly trafficked" during optimal network performance. This measure could help order the arcs in the network to increase the performance of the green and red sub-tree marking algorithms on the associated meta-graph.

2.4.3 Community Detection

Community detection algorithms analyze networks and propose partitions to organize the network nodes into communities where the connections among community members are plentiful and connections between communities are sparse. For the purpose of network resilience and interdiction, we want to focus on non-overlapping community detection algorithms. Two common algorithms are Louvain (Newman 2018, p. 511), which is one of the most widely used for large networks, and Girvan-Newman, a deterministic algorithm commonly used for social networks (Moon et al. 2014). Intuitively, we can isolate a community by attacking the arcs that connect it to other communities. This idea is the basis for the Girvan-Newman community detection algorithm Girvan-Newman Community Detection Algorithm (GN): "By removing these edges, we separate groups from one another and so reveal the underlying community structure of the graph" (Girvan and Newman 2002, p. 7822).

Girvan and Newman (2002) develop a community detection algorithm for a graph using *edge betweenness centrality* since those edges tend to be between communities rather than internal to a specific community. Their algorithm "progressively [removes] edges from the original graph" in order to "separate groups from one another" (Girvan and Newman 2002, p. 7822). GN has a worst-case run time of $O(m^2n)$.

Arasteh and Alizadeh (2019) critique the GN algorithm as suffering from excessive computational complexity since it requires $O(m^3 + m^3 log m)$ for a weighted graph using Dijkstra's algorithm. They offer a faster community detection algorithm that replaces edge betweenness centrality with edge degree betweenness centrality. For an unweighted graph, Arasteh and Alizadeh (2019) define *edge degree betweenness* as the cardinality of the set of edges that have a direct relation with the given edge. For a weighted graph, the edge degree betweenness is the summation of the weights of the directly-related edges. Ostensibly, the benefit of the Arasteh and Alizadeh technique is that one need not compute shortest paths after an edge is removed; instead the operator can reduce edges' centrality measure by the weight of a removed adjacent edge. In this manner, they substitute an $O(m^2 + m^2 log m)$ operation, Dijkstra's algorithm, with a constant time operation for all affected nodes, an $O(m)$ operation. An additional benefit of the Arasteh and Alizadeh method is that multiple edges can be deleted in each iteration. Although Arasteh and Alizadeh (2019) intend to use multiple edge deletion to remove edges with the same centrality value, this technique could

be adjusted to examine the result of a particular attack combination: How do the centralities among the edges change when different combinations of edges are removed? In other words, this could be a fast way to determine network changes over time instead of evaluating the network performance with a traditional Attacker-Defender routine.

2.4.4 Graph Connectedness

Graph *connectedness* is a way to describe how tightly knit a community is. Diestel (2016) explains that a *k-connected* graph means that at least k vertices must be removed in order to disconnect (separate into at least two components) the graph. For example, a friend group (graph) that is *1-connected* could be split into two separate groups (graphs) of friends with the removal of one individual. This individual is the sole connection between the groups and we would say that the original friend group is not *well-connected*.

2.5 Machine Learning

Machine learning is the process of using computers to create a predictive model and can be accomplished by number of different methods, including neural networks and support vector machines (see Knox 2018, for a general introduction). Since machine learning is frequently used for classification problems, many have implemented machine learning to expose potential interactions between predictors in structured data or nodes in networks. Latouche and Rossi (2015) provide an introduction to supervised and unsupervised machine learning on graphs, and focus on graph clustering and multiple graph representation of a common network.

Hammer and Jain (2004) discuss the extension of machine learning to non-standard data, including functional networks, with the desire to answer the following question: Can we approximate multiple, non-linear functions with only the inputs and outputs? (see p .283 Hammer and Jain 2004, for a discussion on neural network approximations of non-linear operators). Zhang et al. (2017) review network-based machine learning and graph theory algorithms related to oncology. Many techniques model drug-drug, drug-target, and drug-disease as relations for a graph and examine graph connectivity measures, classification methods, and link prediction methods. These techniques assume that “drugs tend to be more effective on target genes within or in the vicinity of a disease module in a molecular

network" and their common goal is determine if existing drugs can be used upon new targets to reduce the financial and temporal cost of oncological drug development (Zhang et al. 2017). These methods may be useful in network shaping if failures are viewed as the "mutated or dysregulated pathways" that are the better characterizations for cancer (Zhang et al. 2017). We can view the presence or absence of cancer as the below or above threshold states and determine which failure(s) are most critical in causing system failure.

Gharehbaghi (2016) discusses the application of neural networks for optimization problems of transportation infrastructure maintenance procedures. Gharehbaghi (2016, p. 1) states that the benefits of neural networks include "the ability to operate [on a] large amount of data sets, implicitly detect complex nonlinear relationships between dependent and independent variables, [and to] detect all possible interactions between predictor variables."

Rao and Rao (2012) estimate the future "workload, service rate and utility gain of the web services", which determine the operating state of web applications, by training a neural network and using back propagation.

These various examples provide the potential to train a machine learning algorithm to approximate the network flow problem in various states in order to "guess" where the boundary should be in the metagraph.

2.6 Thesis Contributions

We seek to create an algorithm that focuses on an efficient means to identify the boundary between states in the metagraph. We use two benchmarks to compare our results: exhaustive enumeration and the Alderson-Carlyle enumeration scheme. One technique optimizes the Alderson-Carlyle enumeration scheme by using graph theoretic measures to order the state space. A second technique builds upon the Alderson-Carlyle enumeration scheme by searching for and then traversing along the boundary, marking associated states along the way. The result is a faster method that determines the boundary in the metagraph to quickly reveal the important states explained in Barrow (2019).

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Model

In this chapter, we review and provide additional details for terminology and notation. We then state the problem and discuss potential solution techniques. Finally, we provide our methodology for solving the problem.

3.1 Problem Statement

Let $H = (X, E)$ be the metagraph of the network $G = (N, A)$, where X and N are the sets of vertices and nodes, respectively, and E and A are the sets of edges and arcs, respectively. Then $\forall x \in X$, $p(x)$ is the solution to the maximum flow problem for the augmented network $G_x = (N, A, x)$, where x represents the operating condition of the arcs in the network. That is,

$$p(x) = \max_{\mathbf{y} \in Y(x)} f(x, \mathbf{y}). \tag{3.1}$$

Given a particular performance threshold, τ, we seek to find the metagraph partition of $X = \{\overline{X}, \underline{X}\}$ that separates the states that perform at or above the threshold from the states that perform below the threshold. In other words, we seek the partition where $\forall \overline{x} \in \overline{X}$, $p(\overline{x}) \geq \tau$, and $\forall \underline{x} \in \underline{X}$, $p(\underline{x}) < \tau$.

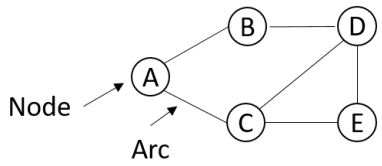
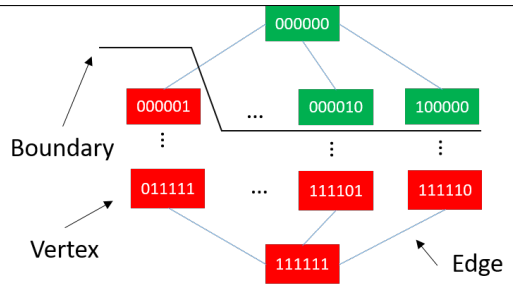
Because we often associate the states $\underline{x} \in \underline{X}$ with *mission failure*, we tend to annotate those states as "failed" and mark their membership by coloring them in red in illustrations. Similarly, for states $\overline{x} \in \overline{X}$, we associate with *mission success*, annotate them as "success" and mark their membership by coloring them green. The process of partitioning the vertices in the metagraph is equivalent to revealing the boundary by evaluating the performance of the states, $p(x)$ for $x \in X$, and determining metagraph coloring.

As a notational convenience, we use a *bitstring* (i.e., a sequence of 0s and 1s) of length m to represent the system state. For a flow network with $m = 6$ arcs, each state in the corresponding metagraph can be represented by the bitstring $b_6 b_5 b_4 b_3 b_2 b_1$, where $b_k \in \{0, 1\}$ indicates if arc k is up or down. The bitstring 000000 indicates that all flow arcs are up, and the bitstring 111111 indicates that all flow arcs are down. Because we assume

that state transitions involve a failure or repair of exactly one component at a time, adjacent vertices in the metagraph will have bitstrings that differ by exactly one bit.

Table 3.1 summarizes terms used to distinguish the elements of a flow network from those in a metagraph.

Table 3.1. Key Taxonomy for Flow Network and Metagraph.

Flow Network	Metagraph
	
A *node* in a flow network represents an origin, destination, or transshipment location.	A *vertex* in a metagraph corresponds to an operating state in a flow network.
An *arc* in a flow network represents the potential for directed movement of a commodity from one node to another. Arc *capacity* is the upper bound on how much flow can pass over the arc. Arc *cost* is per unit of flow over the arc.	An *edge* in a metagraph represents a potential transition from one operating state to another operating state. In the simplest case, an edge represents a binary change in the state of the flow network.
Network *performance* represents some measure of the aggregate flow over the network (e.g., cost). Often, a network has a *threshold* indicating acceptable performance.	The threshold induces a *boundary* in the metagraph that separates the states that have performance above the performance threshold and those below the performance threshold.

Computational Challenge

It should be apparent that the complexity of partitioning X has two aspects: (1) the computation of $p(x)$, and (2) and the number of times the performance must be calculated, which is at most $|X| = 2^m$. As noted, considerable work has gone into identifying efficient algorithms for solving maximum flow problems. One of the most efficient computations of $p(x)$ uses the "Excess scaling algorithm" with run time of $O(nm + n^2 \log U)$ (Ahuja et al. 1993). Because these algorithms are known to be relatively efficient, most of the challenge

here is in identifying the boundary with the fewest possible evaluations $p(x)$ for $x \in X$.

3.2 Potential Solution Techniques

We discuss five potential solution techniques. To compare them, we use the simple flow network in Figure 3.1 where arc capacities are labeled and we aim to maximize the flow from node $n01$ to node $n05$.

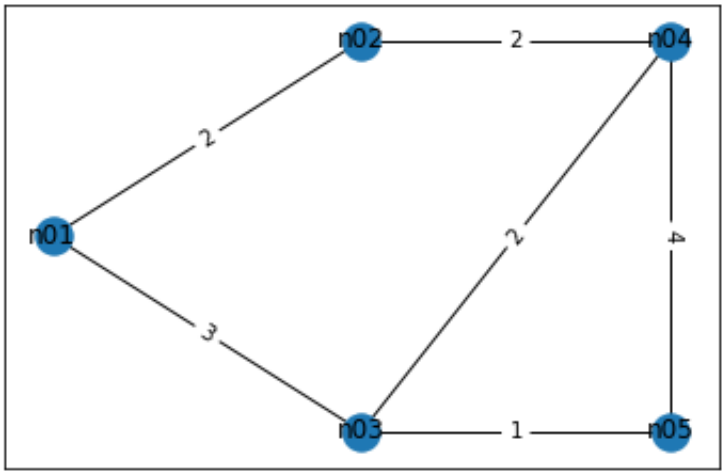


Figure 3.1. Toy Flow Network. This network consists of five nodes and six arcs. The arc capacities are provided on each arc and the maximum flow problem is to solve the maximum flow from node $n01$ to $n05$.

3.2.1 Exhaustive Enumeration

The simplest but most computationally expensive way to partition the metagraph is to conduct exhaustive enumeration, i.e., compute the exact solution to the maximum flow problem for each state in the metagraph.

Figure 3.2 shows the metagraph for the Toy network after conducting exhaustive enumeration. For this example, we select $\tau = 1$ and also include the specific state as the node label. We color the vertices based upon the performance of the associated state. A green vertex

represents that the state performs at or above the threshold, while a red vertex represents that the state performs below the threshold.

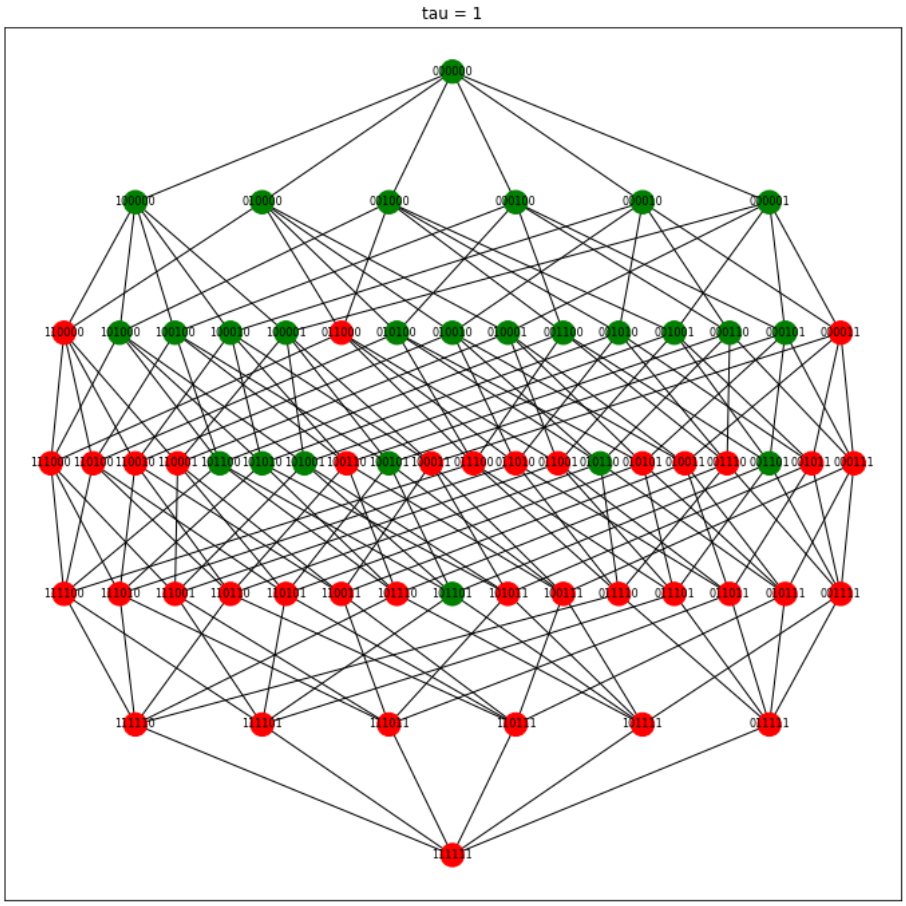


Figure 3.2. Exhaustive Enumeration Metagraph - Toy Network. The metagraph shows which states perform above and below the performance threshold $\tau = 1$. It is visually apparent that the simple flow network with 6 edges has exponentially more states that need $2^6 = 64$ evaluations in this case.

3.2.2 "Smart" Enumeration

Given the computational challenges with exhaustive enumeration, we seek another approach to partitioning the metagraph that reduces the number of states that are evaluated. As noted by Alderson and Carlyle (2017), "[If we] assume performance decreases monotonically with each additional failed component, [then] we can use this to (aggressively) reduce the number of scenarios that we need to evaluate."

Specifically, if we know for a given state $\underline{x}$ that $p(\underline{x}) < \tau$ then any state x' that has the same down arcs (1s in the bitstring), but also additional down arcs, will also have the property $p(x') \leq p(\underline{x}) < \tau$. Accordingly, if we can mark $\underline{x}$ as red then we can also mark x' as red.

We can define a *search tree* on the metagraph as follows. Following the convention in Alderson and Carlyle (2017), for any vertex in the metagraph, "Each child increases the number of damaged arcs (i.e., a superset), partitioned by next arc attacked in sequence (extensions)." We refer to the set of down arcs in a given state as the *down set*. Thus, starting with the completely up state (i.e., a bitstring of all 0s), the sub-tree of each vertex represents *all* down sets containing that vertex's down set (completions). The key feature for this search tree is that if any vertex can be marked red, *then its entire sub-tree can also be marked red.* Figure 3.3 illustrates.

Children in this search tree have only one predecessor vertex, so there is only one path from the root node (e.g., '000000' in Figure 3.3) to any other vertex. Because the connectivity and orientation of this search tree is defined by increasing damage to the system, we refer to this as the "red search tree" or that "the tree grows downward." Moreover, we often say that the vertices in the sub-tree of a vertex are its "downward children".

Enumeration on this "red search tree" becomes straightforward. Starting with the root node, evaluate the performance of the vertex (by solving the max flow problem for the corresponding state). If the performance is above the threshold, mark it green and nominate its children for future evaluation. If the performance is below the performance threshold, then mark the vertex and its entire sub-tree red. This process continues until all vertices are marked. We refer to this enumeration as "red sub-tree marking."

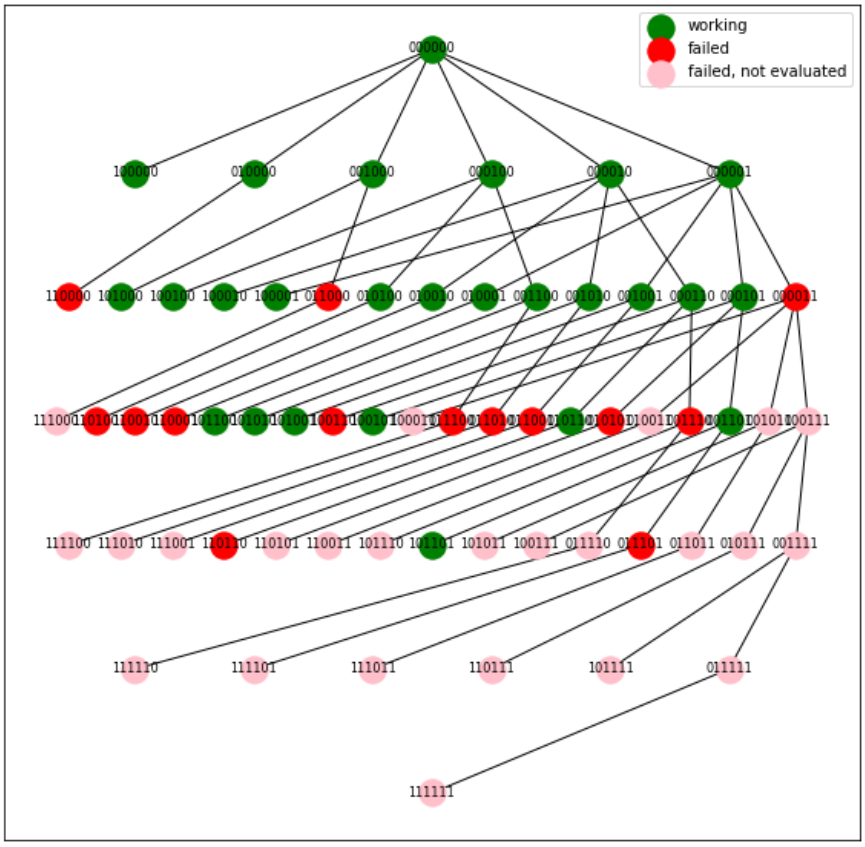


Figure 3.3. "Smart" Enumeration Metagraph - Toy Network, Downward Direction. The pink nodes represent the states that are known to perform below the threshold because of the monotonic assumption and do not need to be evaluated. In this example, 25 nodes did not have to be evaluated because of this assumption, while the remaining 39 nodes did.

This logic works in the other direction as well. If we know for a given state $\overline{x}$ that $p(\overline{x}) \geq \tau$ then we also know that any state x'' that has the same up arcs (bits with value 0) but also additional up arcs will also have the property $p(x'') \geq p(\overline{x}) \geq \tau$. In other words, if we can mark $\overline{x}$ as green then we can also mark x'' as green.

We can create a corresponding "green search tree" and apply this process in the other direction (see Figure 3.4). Because the connectivity and orientation of this search tree is defined by increasing repair to the system, we say that "the tree grows upward." Moreover, we often say that the vertices in the sub-tree of a vertex are its "upward children." We refer to this enumeration as "green sub-tree marking."

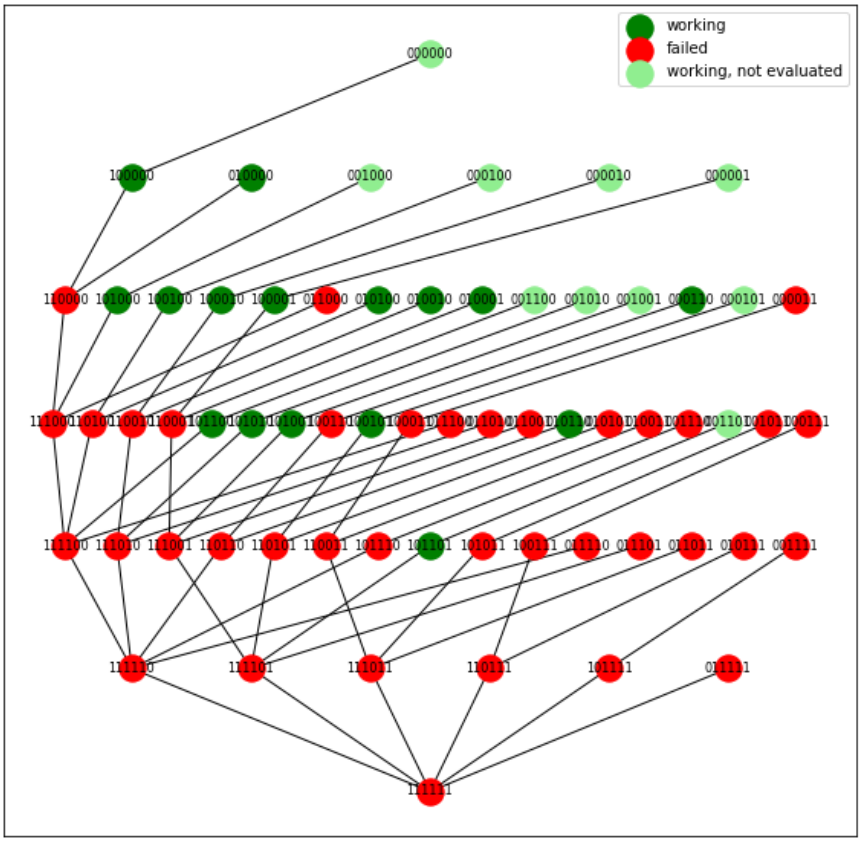


Figure 3.4. "Smart" Enumeration Metagraph - Toy Network, Upward Direction. The light green nodes represent the states that are assumed to perform above the threshold because of the monotonic assumption. In this example, 10 nodes did not have to be evaluated because of this assumption, while the remaining 54 nodes did.

Computational Issues

It is important to note that the efficiency of this "smart enumeration" technique depends on the characteristics of the underlying flow networks as well as the order in which the states are evaluated. If the ordering of the arcs in the bitstring results in the early discovery of red vertices, then a larger number of vertices can be pruned. Considering this in the context of shortest path problems, Clark (2017) provides a reordering algorithm that maximizes the pruning for the red sub-tree.

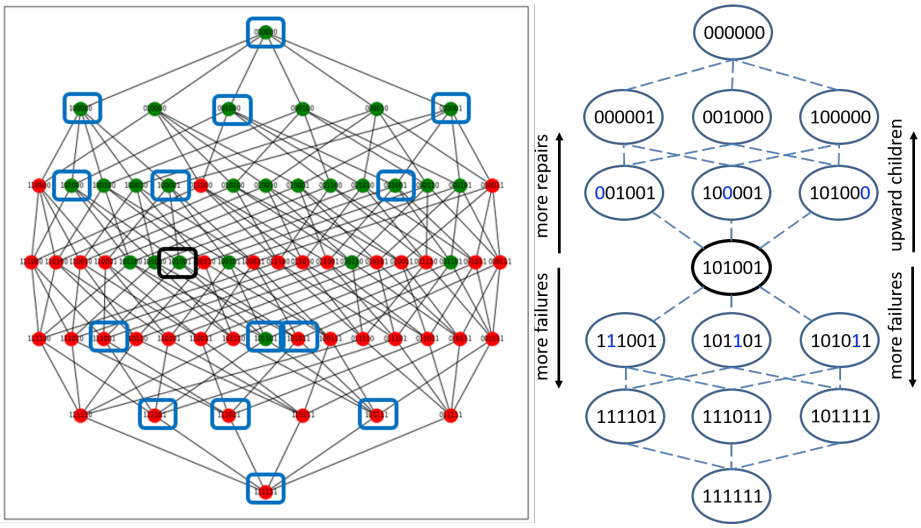


Figure 3.5. Structure of Children In Metagraph. This figure shows how we conceptualize upward and downward children in the metagraph and their relationship to the vertex of interest. The current vertex is highlighted with a black box and both upward and downward children are highlighted with blue boxes. The relationship of which arcs are up and down is shown on the right with a blue digit showing a change from the current vertex.

3.2.3 Boundary Search and Traverse

We introduce a *Boundary Search and Traverse Algorithm (BST)* that employs similar techniques to the "smart" enumeration with its assumption of network performance monotonicity, but it focuses on finding the boundary then evaluating adjacent states that may be on the opposite side of the boundary. The goal is to evaluate the states adjacent to the boundary and thus to minimize the number of evaluated states that are not adjacent to the boundary, thereby reducing computational complexity.

As presented in **Algorithm 1**, BST begins with a flow network and then builds the corresponding metagraph of vertices to be marked. We manage the vertices that are nominated to be evaluated via a First-In First-Out (FIFO) queue. Initially, the queue contains only a single *start vertex*, which for the Toy Flow Network in Figure 3.1 we set as '101010'.

Then, until all the metagraph vertices are marked, BST removes a vertex from the queue and evaluates the performance of its corresponding state (using the Edmonds-Karp algorithm). If the performance is greater than or equal to the performance threshold, then the vertex (and all of its upward children) are marked green. Adjacent vertices in the red direction are added to the queue. If the performance of the state is less than the performance threshold, then the vertex (and all of its downward children) are marked red, while its adjacent vertices in the downward direction are added to the queue.

Algorithm 1: Boundary Search and Traverse (BST) Algorithm

Result: Given a flow network and a performance threshold (*tau*), create the corresponding metagraph and mark all its vertices as red or green.

Given a flow network with m arcs, instantiate `metagraph` with 2^m vertices;
Set all vertices in `metagraph` as *unmarked*;
Set *num_marked* $\leftarrow$ 0;
Initialize empty `queue`;
Identify *start vertex* and add to `queue`;
while *num_marked* < 2^m **do**
    nextstate $\leftarrow$ remove next vertex from `queue`;
    $state\_solution \leftarrow$ Edmonds_Karp(*nextstate*);
    if *state_solution* >= *tau* **then**
        mark *nextstate* as *green*;
        mark upward children of *nextstate* as *green*;
        add adjacent states in downward direction to `queue`;
    end
    if *state_solution* < *tau* **then**
        mark *nextstate* as *red*;
        mark downward children of *nextstate* as *red*;
        add adjacent states in upward direction to `queue`;
    end
end

Figure 3.6 illustrates the first few steps of BST for our Toy Network. The final marked metagraph is illustrated in Figure 3.7.

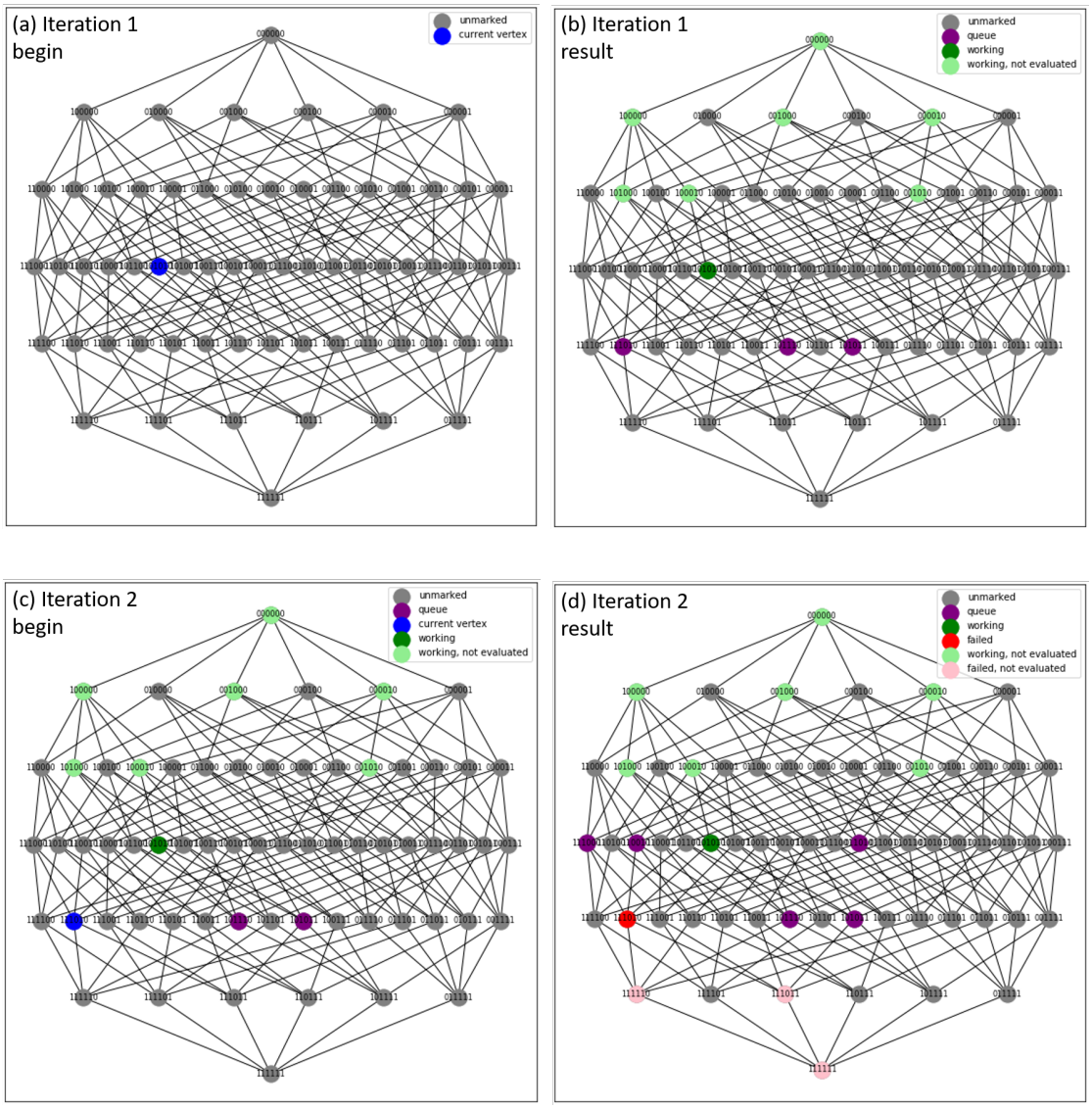


Figure 3.6. Algorithmic Progression Of BST. Figure 3.6(a) shows the initialization where the start vertex is '101010'. We evaluate the maximum flow problem for that state and determine the solution to the maximum flow problem is above the threshold of 1, so we mark the vertex green, as shown in the Figure 3.6(b). The upward children are then marked light green, and the adjacent vertices in the downward direction are added to the queue, shown in light blue. We then pop one vertex off of the queue to evaluate for the next iteration, shown in purple in Figure 3.6(c). Figure 3.6(d) shows that vertex '111010' performs below the threshold, so it is marked red, its downward children are marked pink, and adjacent upward vertices are added to the queue. This process continues until all vertices are marked.

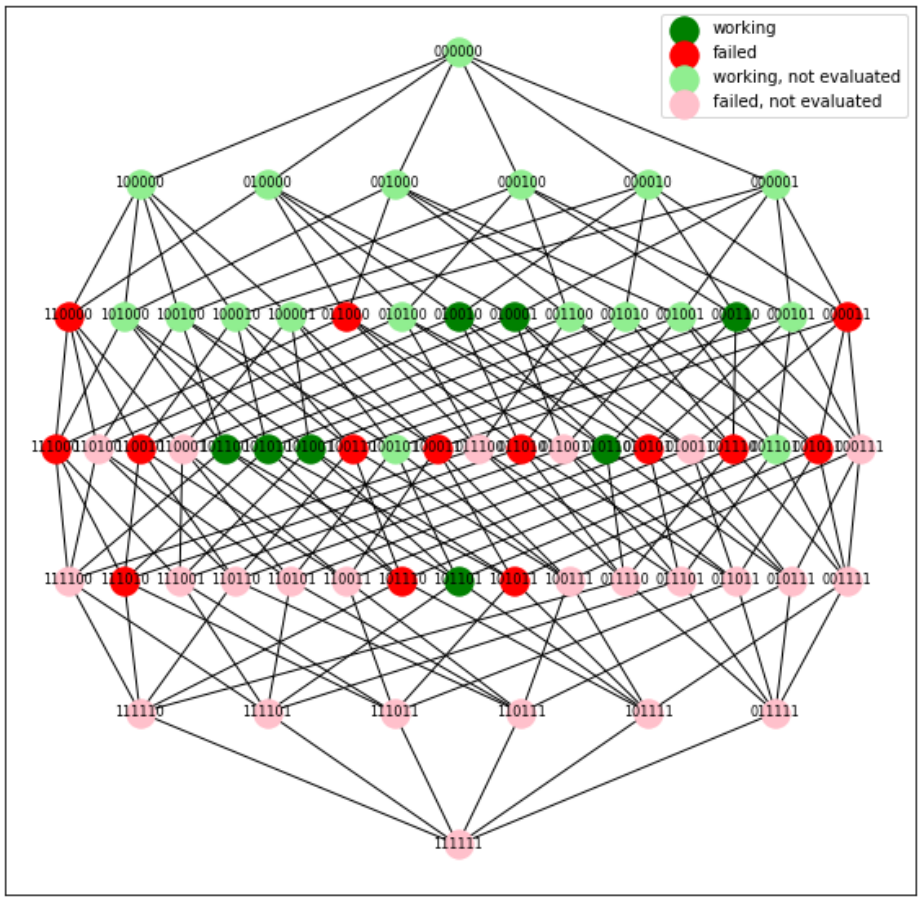


Figure 3.7. BST Enumeration Metagraph - Toy Network. The light green nodes represent the states that are assumed to perform above the threshold because of the monotonic assumption while the pink nodes are assumed below the threshold. In this example, 42 nodes did not have to be evaluated because of this assumption while the remaining 22 nodes did.

Implementation Issues

A key difference between BST and the "smart enumeration" techniques in Section 3.2.2 is that BST does not restrict its marking of children to sub-trees. Rather, we mark *all children* in either the upward or downward direction whenever possible.

We implement BST in three different ways: (1) a Python dictionary for the metagraph and a list for the queue; (2) a Python dictionary for the metagraph and a set for the queue; (3) an n-dimensional array using the NumPy library (Harris et al. 2020) for the metagraph and

a Python list for the queue. Our choice of implementation for BST dictates the memory required, the marking speed, and the marking efficiency.

For example, in the case where we represent the states in a NumPy n-dimensional array, we can identify the children of a vertex using NumPy's built-in filter operations. To determine all of the children in the downward direction, not just the adjacent vertices, we conduct an `AND` operation between the current state and the full array. We then filter then result for the new values that match the current state. The positions that match are exactly all of the positions for children in the downward direction for the original array. The same method can be used with an `OR` operation to determine the upward children. The logic is as follows: 101010 `AND` 111111 = 101010, so 111111 is a downward child of 101010; 101010 `OR` 000000 = 101010, so 000000 is an upward child of 101010. The completely down and completely up states are representative of all the downward and upward children of the state 101010 and this operation can be done simultaneously over the full 2^m array. This implementation marks states quickly.

3.2.4 Network Science Heuristics

Graph theoretic measures can be used to characterize the structure of the network. One potential use for this information would be to focus on which arcs may be more "important" than others and focus on states that represent failures on those arcs, thereby reducing the number of $p(x)$ calculations. Alternatively, these measures could be used as a heuristic to approximate the performance of the network, thereby reducing the complexity of each performance calculation. In general, heuristic measures alone do not do a good job approximating network performance (Alderson et al. 2011), so we need to combine heuristics with additional techniques.

In this thesis, we refer to edge betweenness centrality as *Arc Betweenness Centrality (ABC)* since we compute the measure on the network arcs and not the metagraph edges. In Table 3.2, ABC shows the relative proportion of how many shortest paths in the network pass over the particular arc (Girvan and Newman 2002). In the maximum flow problem, path length is not considered because the problem focuses only on the capacities of the arcs and not the distance. One simple modification to incorporate arc capacities for ABC is to invert the capacities of the arcs to represent that a larger capacity is 'better' than smaller capacity.

Thinking of the flow as a time-based rate, the inversion of the capacity merely represents how long a supply would take to travel along the path. For example, suppose there are two paths from node a to node b, where the first path has $capacity = 1$ and the second path has $capacity = 2$. Assuming equal distance and travel speed along the arc, a commodity of sufficient quantity would take twice as long to move along the first path than the second path.

Table 3.2. Arc Betweenness Centrality - Toy Network. As originally defined, an unweighted Arc Betweenness Centrality may be limited when considering maximum flow problems as opposed to shortest path problems. To account for the effect of capacities on the maximum flow solution, it may be useful to weight the arcs based on their capacities.

Arc	$u_{i,j}$	ABC (unweighted)	ABC (weight = $\frac{1}{u_{i,j}}$)
(n01,n02)	2	2	2
(n01,n03)	3	3	4
(n02,n04)	2	3	2
(n03,n04)	2	2	4
(n03,n05)	1	2	0
(n04,n05)	4	2	4

We introduce the *Arc Eigenvector Centrality (AEC)* by computing the eigenvector centrality of the arc adjacency matrix, as a variant of the eigenvector centrality (page rank) in the edge-adjacency matrix (Mishkovski et al. 2010). The AEC computes the relative importance of an arc in a recursive manner by computing the leading eigenvector of the arc adjacency matrix. Table 3.3 shows the value of the arcs in the flow network in Figure 3.1.

Since the arc adjacency matrix does not account for the capacities of the arcs, we also provide two modified centrality measures: AEC multiplied by the capacity, to provide a crude incorporation of the arc capacities, and a modified AEC, where we add the arc capacities to the main diagonal of the arc adjacency matrix. In Table 3.3, we also provide the relative arc ranking in parentheses.

Table 3.3. Arc Eigenvector Centrality - Toy Network.

Arc	$u_{i,j}$	AEC	AEC $\times$ $u_{i,j}$	Modified AEC
(n01,n02)	2 (3)	0.575 (6)	1.151 (5)	0.290 (6)
(n01,n03)	3 (2)	0.896 (4)	2.690 (2)	0.538 (3)
(n02,n04)	2 (3)	0.896 (4)	1.793 (4)	0.535 (4)
(n03,n04)	2 (3)	1.217 (1)	2.435 (3)	0.688 (2)
(n03,n05)	1 (6)	1.0 (2)	1.0 (6)	0.474 (5)
(n04,n05)	4 (1)	1.0 (2)	4.0 (1)	1.0 (1)

3.3 Metric for Algorithmic Performance

To measure the relative performance associated with any technique, we use two methods based on network size. First, we consider the number of vertices in the metagraph that have to be evaluated (with fewer being more efficient). Second, we also consider the usual computational complexity measures of runtime and memory usage (again, with less being more efficient).

We note that the ordering of arcs from the original flow network, and subsequent ordering of the associated bitstring for vertices, can greatly impact the performance of each algorithm. This is most apparent for the red sub-tree and green sub-tree marking methods, whose enumeration trees are dictated by the order of the bitstring. As noted, Clark (2017) provides an algorithm for reordering the arcs that maximizes the pruning for the red sub-tree in the context of a shortest-path network flow problem. But how much can this performance vary?

For small networks, we can conduct exhaustive enumeration of all possible arc orderings, test the algorithm on the resulting metagraph, and then determine the proportion of red vertices that are pruned. In large networks, the number of possible state orderings is $m!$, so we take a random sample of possible orderings and compute the number of pruned vertices for each. Both techniques result in a distribution between the largest number of pruned red vertices (the best performance) and the fewest number of pruned red vertices (the worst performance). Thus, the variability in algorithmic efficiency of a given technique can be assessed.

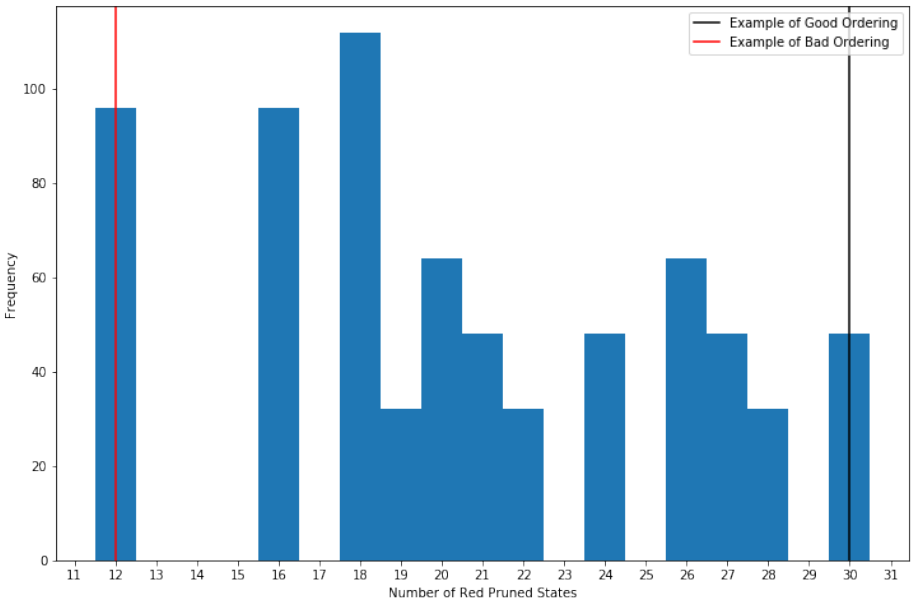


Figure 3.8. Example Histogram for Red Pruning. The histogram shows the amount of red pruning for all possible orderings on the Toy flow network in Figure 3.1.

Figure 3.8 shows a histogram for the number of red pruned states for all possible orderings on the Toy flow network. The black line represents the best possible ordering for the red sub-tree algorithm (resulting in 30 pruned vertices), while the red line represents the worst possible ordering (resulting in only 13 pruned vertices). The goal of a heuristic ordering is to provide good orderings that maximize the amount of pruned states.

In summary, the goal of all the discussed methods is to reduce computational complexity for enumerating the metagraph states. The primary method to accomplish the goal is by minimizing the total amount of states that require evaluation, or, equivalently, by maximizing the number of states that are pruned. The underlying assumption for all of these methods is that any additional computational requirements for the various algorithms is exceeded by the benefits of evaluating fewer metagraph states.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

To evaluate the performance of the different algorithms, we employ them on small example networks to reveal any reliance upon network structure and then validate the performance on a larger network with historical importance.

4.1 Example Networks

First, we compare graph theoretic characteristics of several example networks to show that they are sufficiently different from a structural perspective. Then, we compare the performance of the different methods on the example networks to evaluate the suitability of the methods on different types of networks. The example networks are intended to differ with the exception of having the same maximum flow solution under optimal performance, to ensure comparability.

4.1.1 Comparing the Structure of the Example Networks

We use four example networks, shown in Figure 4.1, to evaluate the performance of the different methods: Simple2, Distribution, Parallel, and Lattice1. All four networks have a maximum flow solution of 9 under optimal conditions. In Simple2 network (Figure 4.1a), the source node is $n01$, the sink node is $n09$, and all other nodes are transfer nodes, nodes that have no supply or demand. In Distribution network (Figure 4.1b), the source node is $n01$ and the sink nodes are all of the leaves: $n02$, $n04$, $n06$, $n08$, $n10$, $n11$, $n13$, $n14$, and $n16$. In Parallel network (Figure 4.1c), the source node is $n01$ and the sink node is $n10$. Finally, in Lattice1 network (Figure 4.1d), the source node is $n01$ and the sink node is $n08$. For all networks, the arc capacities are noted along the arc in the network and range from 1 to 5 units.

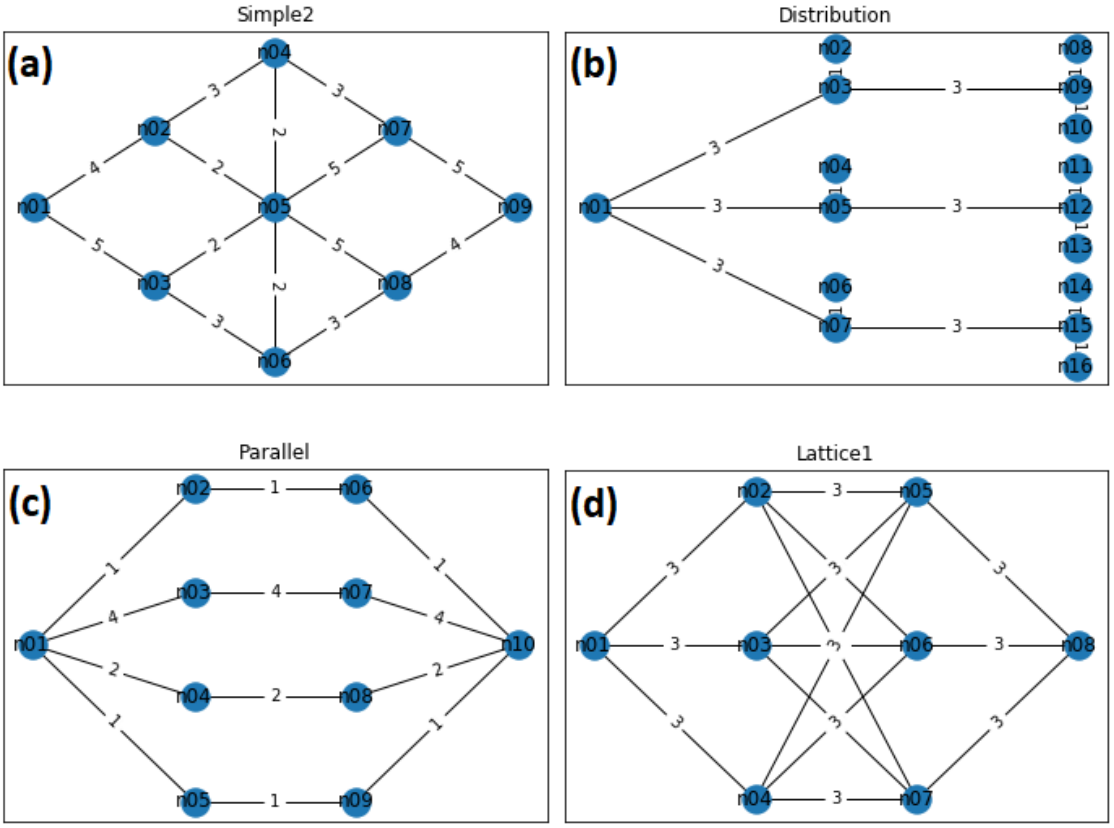


Figure 4.1. Example Networks. Four Example Networks: (a) Simple2, with maximum flow from $n01$ to $n09$; (b) Distribution, with maximum flow from $n01$ to nodes $n02$, $n04$, $n06$, $n08$, $n10$, $n11$, $n13$, $n14$, and $n16$; (c) Parallel, with maximum flow from $n01$, to $n10$; and (d) Lattice1, with maximum flow from $n01$ to $n08$.

To compare the structure of the networks, Table 4.1 shows the number of nodes, arcs, diameter, radius, arc connectivity, and node connectivity, while Figure 4.2 shows the node degree distribution, ABC, $s-t$ subsetted ABC, and AEC.

Table 4.1. Network Comparison - Graph Theoretic Properties. Structural differences among the four example networks can be quantified. These six basic characteristics show that the networks have different attributes.

Network $G = (N, A)$	Simple2	Distribution	Parallel	Lattice1
$\|N\|$ or $order$	9	16	10	8
$\|A\|$ or $size$	14	15	12	15
$diam(G)$	4	6	3	3
$rad(G)$	2	3	3	2
$\lambda(G)$	2	1	2	3
$\kappa(G)$	2	1	2	3

From Table 4.1, we can see that all four networks have similar *order* (number of nodes), and *size* (number of arcs), though the tree structure of the Distribution network, which can be seen visually and since $size = order - 1$, is an obvious difference. Parallel network differs from the others since it is the only example network where $diam(G) = rad(G)$. Finally, Lattice1 differs from the others since $diam(G) = \lambda(G)$.

Figure 4.2 shows additional graph theoretic properties for the four example networks. The vertical axes for all subplots in Figure 4.2 are scaled from zero to the number of nodes or arcs in the network, nodes for the node degree plot and arcs for the three others. For the node degree plots, the horizontal axes range from zero to $|N| - 1$, the maximum possible value in the particular network, both types of ABC plots are normalized from zero to one, and the AEC plots are scaled from zero to four.

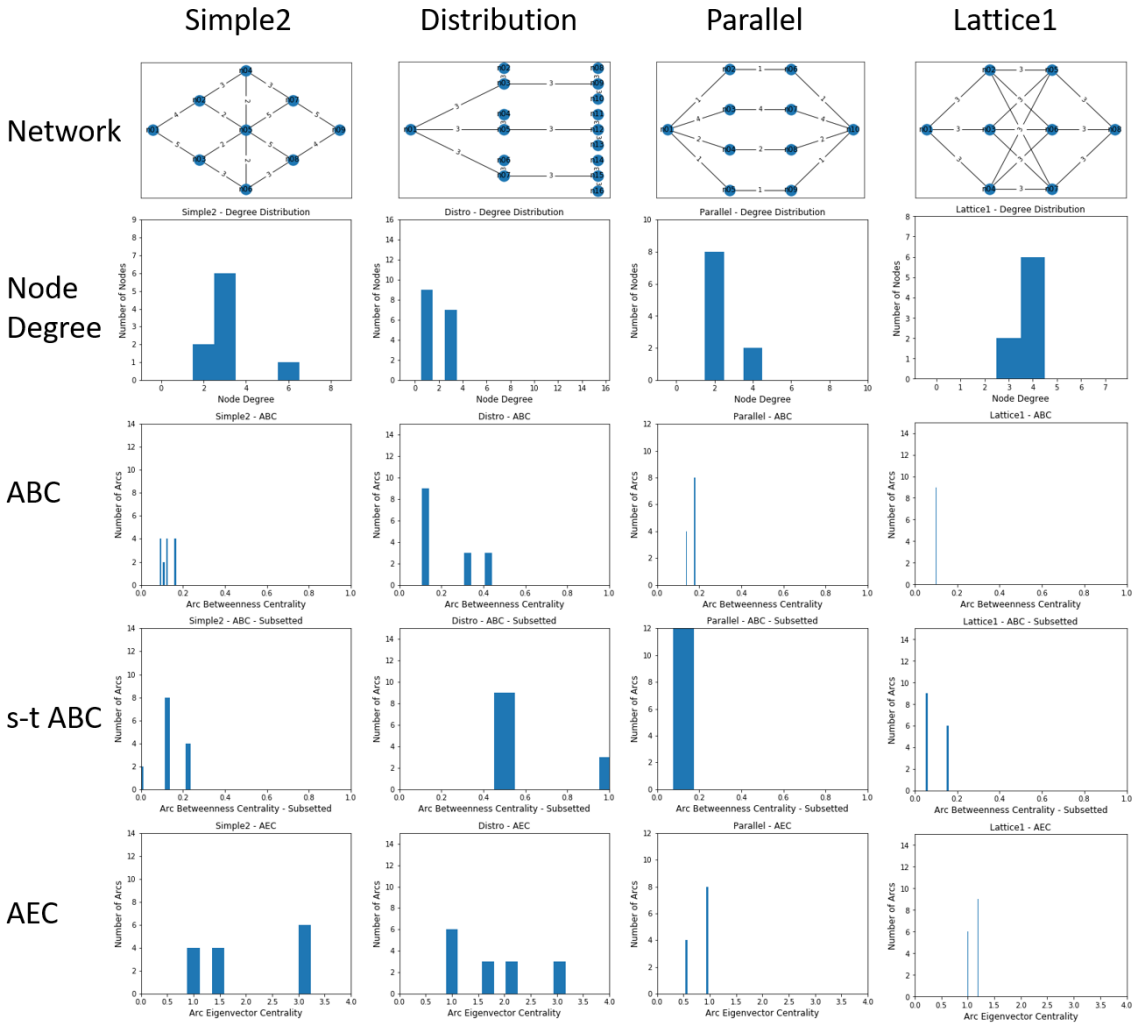


Figure 4.2. Network Comparison - Graph Theoretic Properties.

The main takeaway from Figure 4.2 is that the networks vary in these graph theoretic properties. Also, even though two networks may appear similar for one graph theoretic property, the networks differ in others. This also introduces the idea that different measures may be useful for different networks.

4.1.2 Algorithmic Performance on the Example Networks

For each of the four example networks, we apply three algorithms—red sub-tree marking, green sub-tree marking, and BST—to solve for the boundary between red and green vertices

the corresponding metagraph. We compare the amount of work (measured in terms of the number of states evaluated) for each algorithm, as compared to exhaustive enumeration.

Simple2 Network. The Simple2 network has 14 arcs, which corresponds to $2^{14} = 16,384$ total vertices in the corresponding metagraph.

Figure 4.3 contrasts the performance of the three algorithms to exhaustive enumeration for different performance threshold values (τ). The higher the performance threshold, the easier it is to interdict the network into a red state. Figure 4.3a shows the amount of red and green states that need to be evaluated for exhaustive enumeration. The balance between red and green states depends on the particular threshold and Figure 4.3a shows that the amount of green states ranges from as high as 5,298 to as low as 23, while the remainder of the 16,384 states are red. The importance of this balance manifests in the efficiency achieved by the various algorithms. The other three sub-figures represent the algorithmic performance of the red sub-tree (Figure 4.3b), green sub-tree (Figure 4.3c), and BST algorithms (Figure 4.3d). As in chapter 3, the pink portions of the stacked bar chart represent the red states that did not need to be evaluated and the light green portions represent the green states that did not need to be evaluated. In Figures 4.3b-d, the numbers on the bars show the amount of states that were not evaluated, whether pink, light green, or both. The higher these numbers, the greater the computational savings of the technique.

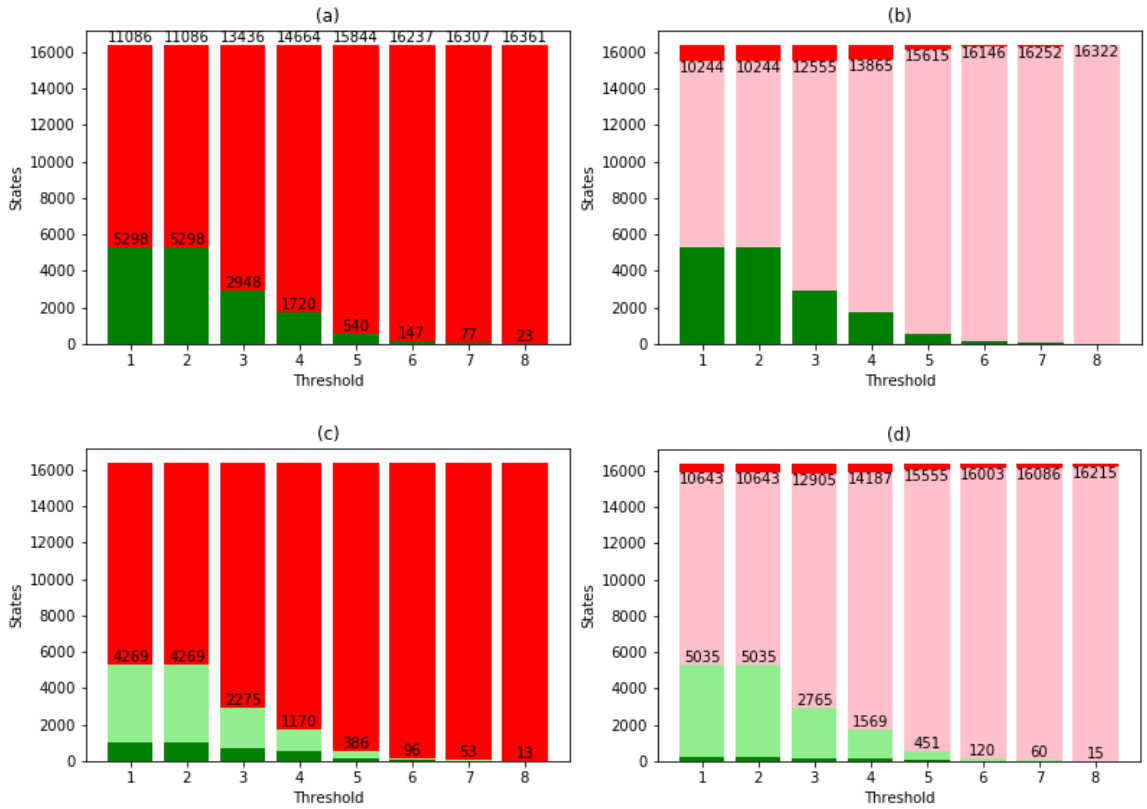


Figure 4.3. Algorithm Performance Comparison - Simple2 Network. Figure 4.3a shows how many red and green states are evaluated by conducting exhaustive enumeration. Figure 4.3b highlights the savings from the red sub-tree method, ranging from 10,244 to 16,322. Figure 4.3c highlights the savings from the green sub-tree method, ranging from 13 to 4,269. Figure 4.3d highlights the savings from BST, ranging from 15,670 to 16,230.

From the plots in the Figure 4.3, it is obvious that the performance of the different algorithms is sensitive to the particular threshold chosen in the network. BST (Figure 4.3d) requires the fewest evaluations of the maximum flow problem among all considered algorithms for the Simple2 network. This result is consistent among all the example networks. On the Simple2 network, the red sub-tree algorithm (Figure 4.3b) outperforms the green sub-tree algorithm (Figure 4.3c), presumably because there are more red states than green states across all relevant thresholds (thresholds of 0 and 9, the maximum flow solution under optimal conditions, are considered trivial for our purposes).

Distribution Network. The Distribution network has 15 arcs, which corresponds to $2^{15} = 32,768$ total vertices in the corresponding metagraph.

In Figure 4.4, we compare the amount of red and green states that need to be evaluated for the red sub-tree, green sub-tree, and BST algorithms on the Distribution network. For low performance thresholds, the Distribution network has more green states than red states.

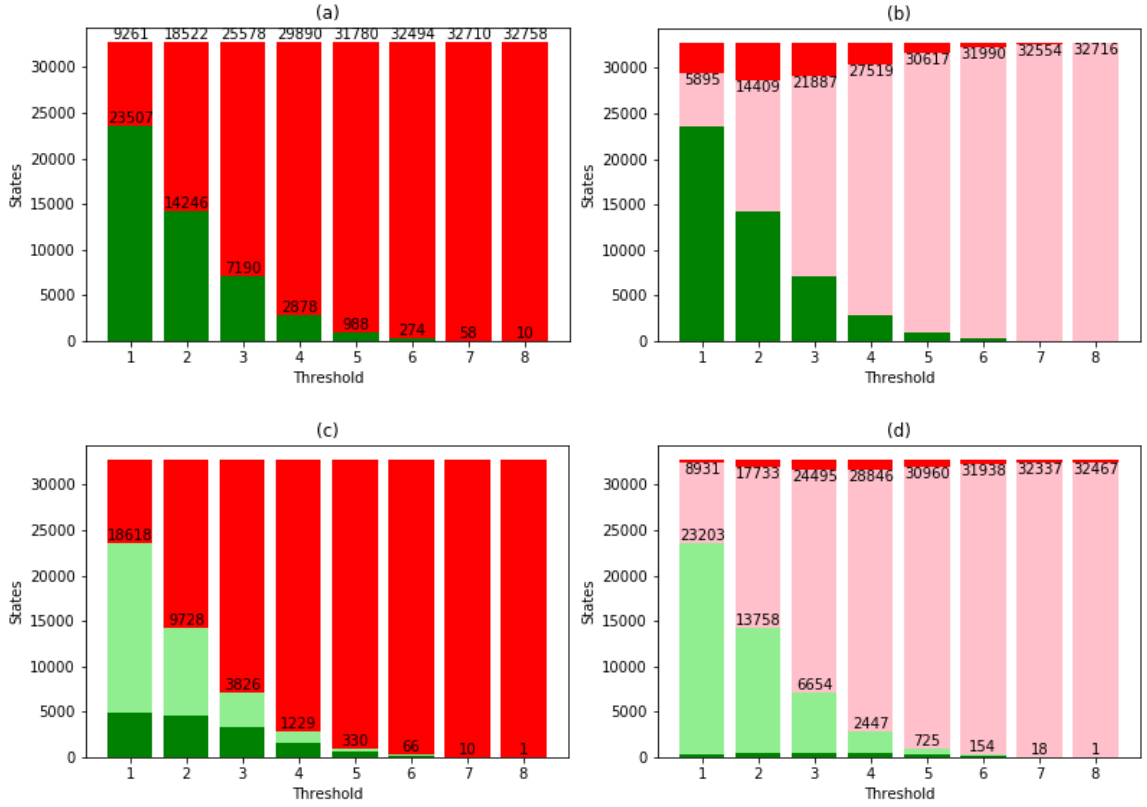


Figure 4.4. Algorithm Performance Comparison - Distribution Network. Figure 4.4a shows how many red and green states are evaluated by conducting exhaustive enumeration. Figure 4.4b highlights the savings from the red sub-tree method, ranging from 5,895 to 32,716. Figure 4.4c highlights the savings from the green sub-tree method, ranging from 1 to 18,618. Figure 4.4d highlights the savings from BST, ranging from 31,149 to 32,468.

The relative performance of the algorithms on the Distribution network differ from the Simple2 network only between the red and green methods on low thresholds, $\tau = 1$. This is most likely because the Distribution network has more green states than red states for

the low threshold value. As in the Simple2 network, the BST requires the fewest evaluated states.

Parallel Network. The Parallel network has 12 arcs, which corresponds to $2^{12} = 4,096$ total vertices in the corresponding metagraph.

The Parallel network is red heavy across all thresholds, like the Simple2 network, even though the two networks differ in their graph theoretic properties.

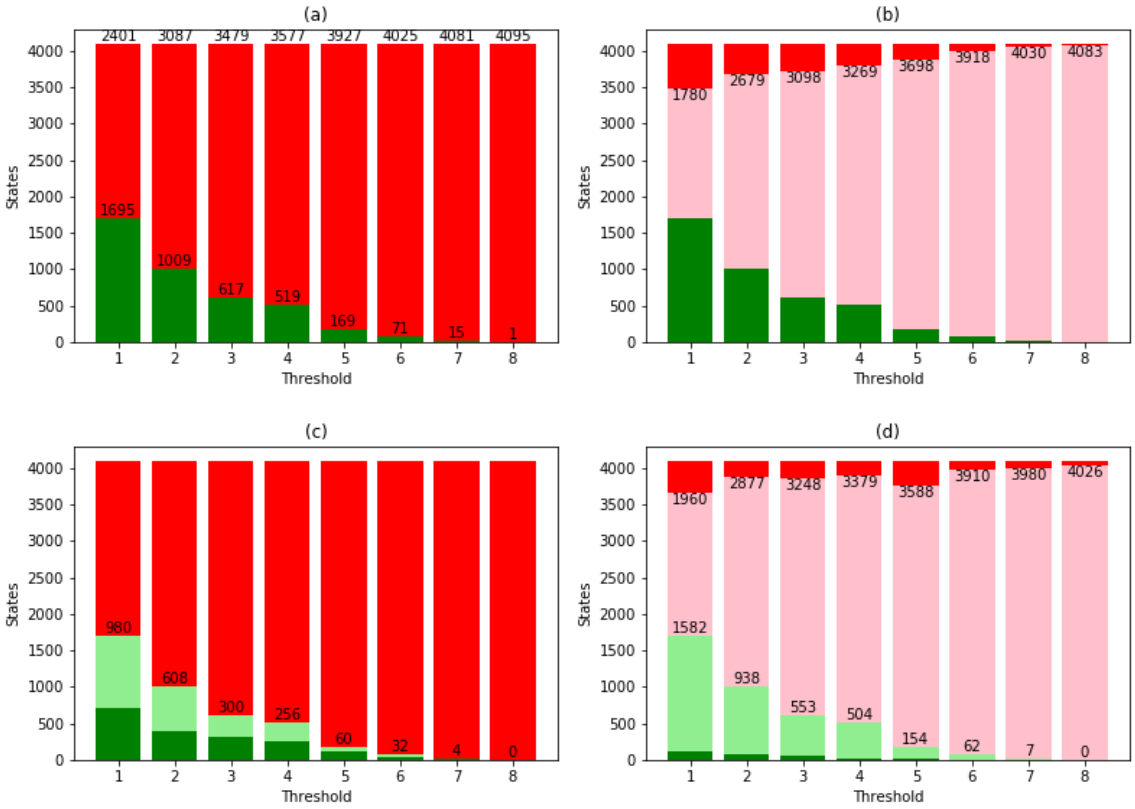


Figure 4.5. Algorithm Performance Comparison - Parallel Network. Figure 4.5a shows how many red and green states are evaluated by conducting exhaustive enumeration. Figure 4.5b highlights the savings from the red sub-tree method, ranging from 1,780 to 4,083. Figure 4.5c highlights the savings from the green sub-tree method, ranging from 0 to 980. Figure 4.5d highlights the savings from BST, ranging from 3,542 to 4,026.

The red marking outperforms the green marking algorithm on the Parallel network, pre-

sumably because of the relative proportion of red to green states. As will the other example networks, BST requires fewer evaluated states. Finally, we examine the performance of the algorithms on the Lattice1 network.

Lattice1 Network. The Lattice1 network has 15 arcs, which corresponds to $2^{15} = 32,768$ total vertices in the corresponding metagraph.

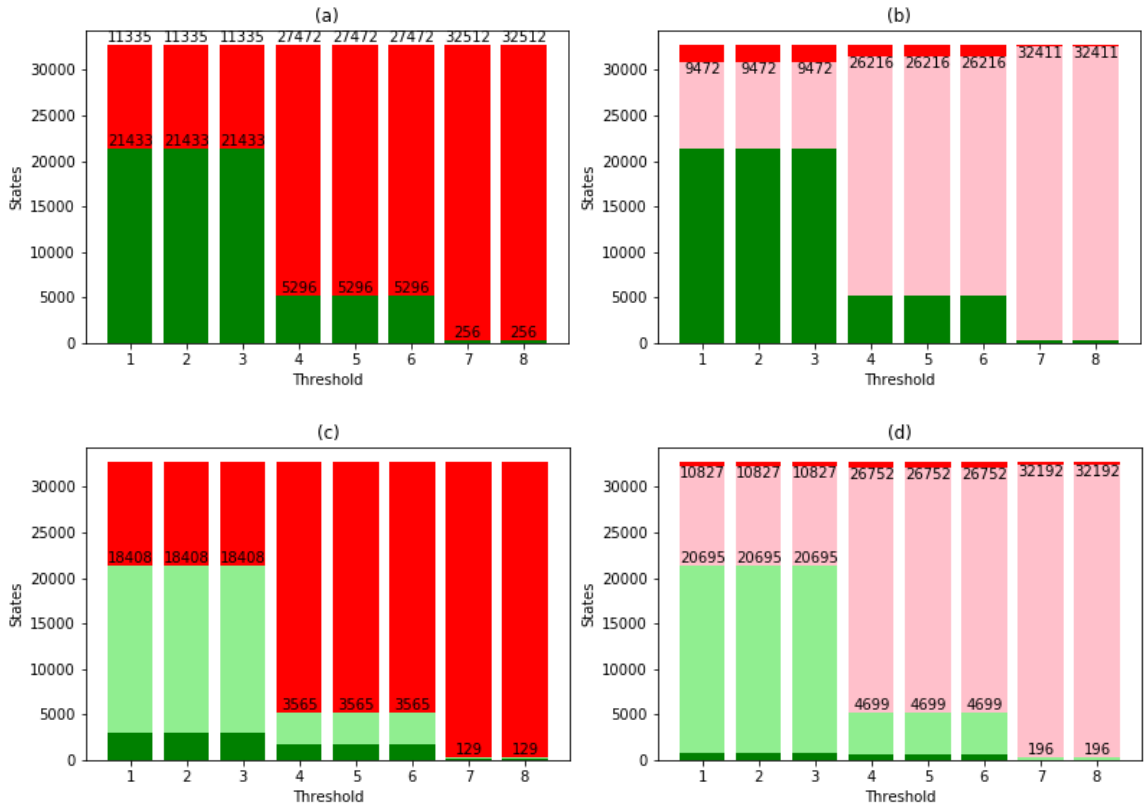


Figure 4.6. Algorithm Performance Comparison - Lattice1 Network. Figure 4.6a shows how many red and green states are evaluated by conducting exhaustive enumeration. Figure 4.6b highlights the savings from the red sub-tree method, ranging from 9,472 to 32,411. Figure 4.6c highlights the savings from the green sub-tree method, ranging from 129 to 18,408. Figure 4.6d highlights the savings from BST, ranging from 31,522 to 32,388.

The relative performance of the algorithms on the Lattice1 network are very similar to those on the Distribution network, most likely because the threshold values of one through three that are green heavy and threshold values above three are red heavy.

4.1.3 Algorithmic Improvement Using State Space Ordering

To determine how well the red sub-tree and green sub-tree marking methods could possibly perform, we compare the performance of the methods using various state space orderings to possible orderings. Figure 4.7 shows the performance of various heuristic orderings using the red tree algorithm and compares them to a random sample of $1,000$ possible orderings, without replacement. The motivating question is whether we can determine a heuristic method to consistently improve the performance of the red, green, and BST methods.

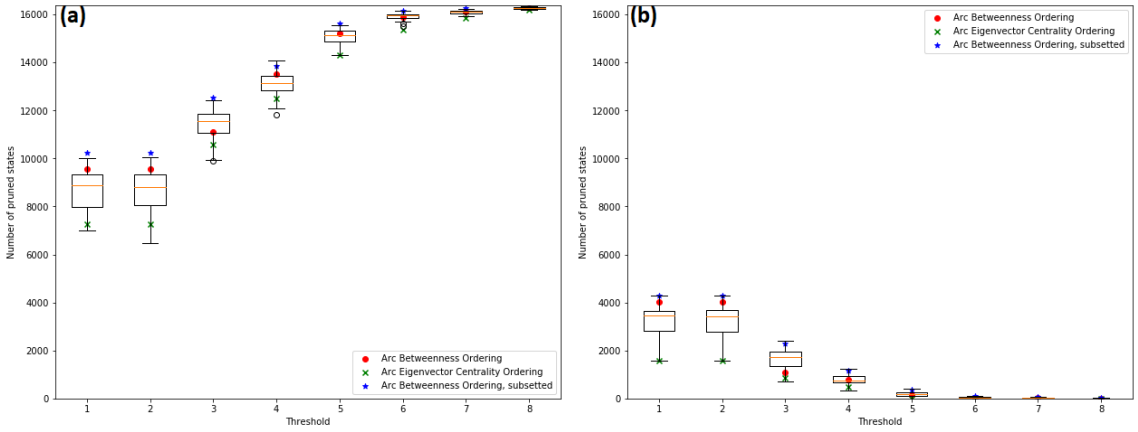


Figure 4.7. Heuristic Ordering Performance - Simple2 Network. (a) The performance of various state space orderings while using the red sub-tree method compared to the performance distribution of the red sub-tree method for a sample of possible state space orderings. (b) The performance of various state space orderings while using the green sub-tree method.

Figure 4.7 is an extension of the histogram presented in Figure 3.8, with a distribution for different threshold levels. In general, higher performance thresholds for the underlying maxflow problem make red sub-tree marking more effective (Figure 4.7a), while lower performance thresholds make green sub-tree marking more effective (Figure 4.7b). For both methods, we observe the most variability in performance by ordering for low threshold values. Moreover, for low threshold values, we observe that red sub-tree marking is more effective at pruning vertices.

Figure 4.7 also identifies the way in which particular ordering schemes benefit the performance of each method. For the Simple2 network, we can see in Figure 4.7 that the $s - t$ subsetted ABC ordering tends to perform well in conjunction with both the red sub-tree and

green sub-tree marking methods.

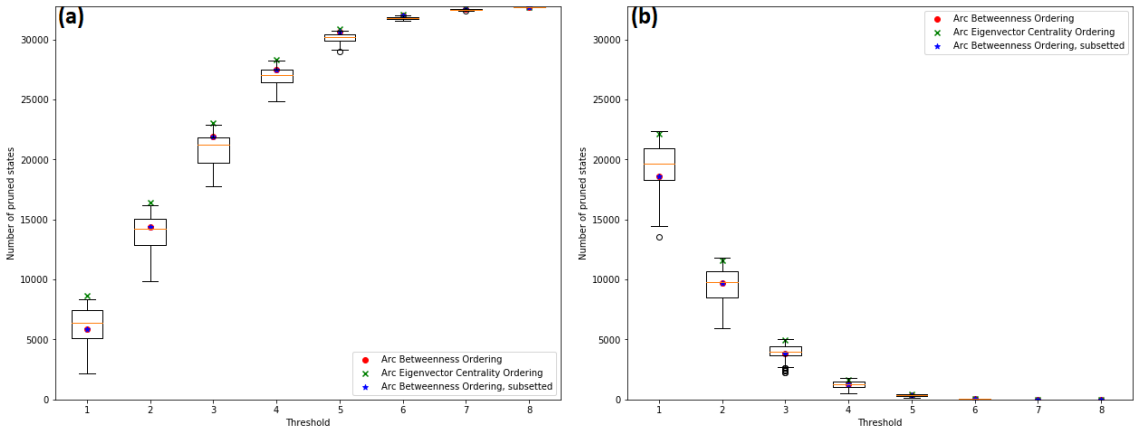


Figure 4.8. Heuristic Ordering Performance - Distribution Network. (a) The performance of various state space orderings while using the red sub-tree method compared to the performance distribution of the red sub-tree method for a sample of possible state space orderings. (b) The performance of various state space orderings while using the green sub-tree method.

Figure 4.8 shows the impact of different orderings for sub-tree marking algorithms when applied to the Distribution network. For small threshold values, we observe that green sub-tree marking results in more pruned vertices (Figure 4.8b) than the red sub-tree method (Figure 4.8a). We also observe that AEC ordering performs well for both marking methods, while the two ABC orderings perform about as well as a random ordering.

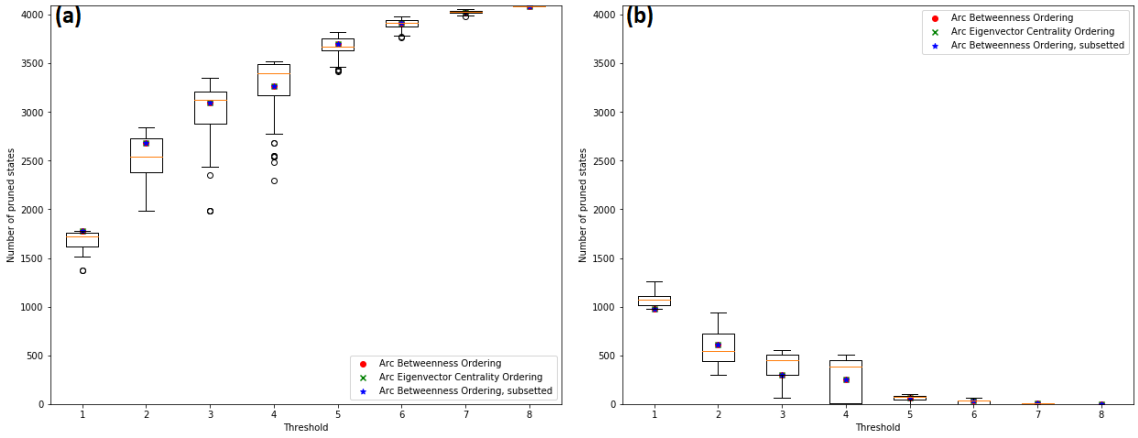


Figure 4.9. Heuristic Ordering Performance - Parallel Network. (a) The performance of various state space orderings while using the red sub-tree method compared to the performance distribution of the red sub-tree method for a sample of possible state space orderings. (b) The performance of various state space orderings while using the green sub-tree method.

Figure 4.9 shows the impact of different orderings for sub-tree marking algorithms when applied to the Parallel network. For low threshold values, we observe that red sub-tree marking (Figure 4.9a) is more effective at pruning vertices than the green sub-tree marking (Figure 4.9b). Moreover, we observe that all of the heuristic orderings perform about the same and tend to perform about the same as the random orderings.

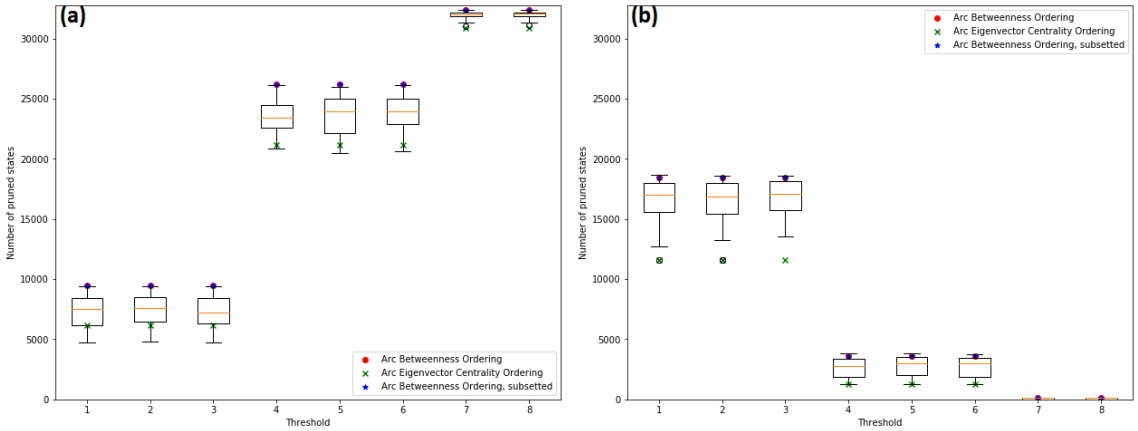


Figure 4.10. Heuristic Ordering Performance - Lattice1 Network. (a) The performance of various state space orderings while using the red sub-tree method compared to the performance distribution of the red sub-tree method for a sample of possible state space orderings. (b) The performance of various state space orderings while using the green sub-tree method.

Figure 4.10 shows the impact of different orderings for sub-tree marking algorithms when applied to the Lattice1 network. For small threshold values, we observe that green sub-tree marking (Figure 4.10b) results in more pruned vertices than red sub-tree marking (Figure 4.10a). We also observe something similar to the Simple2 network in that the two ABC orderings perform well with both the red and green marking methods, as shown in Figure 4.7.

In summary, networks with structural similarity to the Distribution network may be best suited with the AEC ordering, while networks similar to the Simple2 and Lattice1 may take advantage of the two ABC orderings, though the $s - t$ subsetted alternative performing as well or better than the other. For Parallel-like networks, there is no performance difference among these heuristic orderings.

4.2 Soviet Railroad Network

The history of the maximum flow problem can be traced back to the seminal work by Harris and Ross (1955) and their evaluation of the Soviet railroad network. Figure 4.11 shows the network as depicted in the original work by Harris and Ross (1955) and Figure 4.12 shows our depiction as a network flow model, where railroad lines are represented by arcs and railroad intersections are represented by nodes.

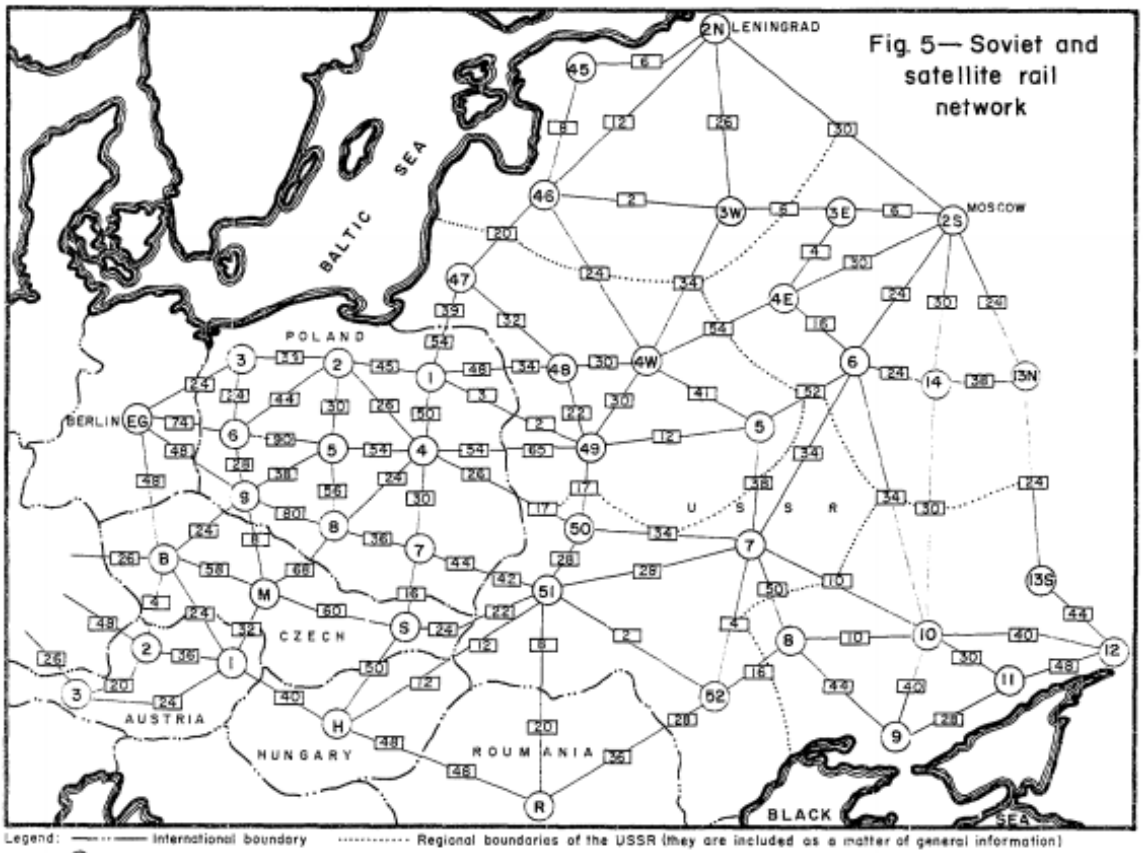


Figure 4.11. Soviet Rail Network - Original Depiction (Harris and Ross 1955).

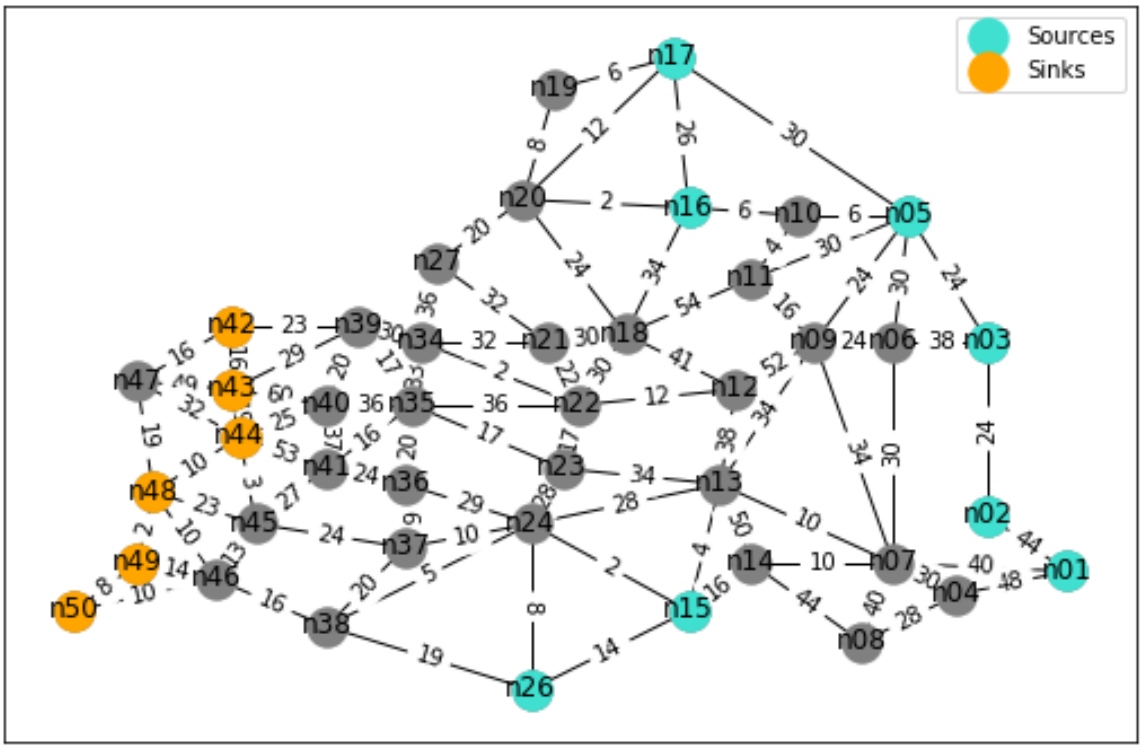


Figure 4.12. Soviet Rail Network - Network Flow Model. Arcs are labelled by their capacity. The goal is to maximize the flow from source nodes to sink nodes.

This flow network has 43 nodes and 94 arcs, and the maximum flow for the original network is 163. Unfortunately, the size of the resulting metagraph with 2^{94} vertices creates a significant computational challenge. We conduct our numerical experiments on a workstation with an Intel(R) Xenon(R) Gold 6230 CPU with a 2.10GHz processor and 128GB of RAM as the testing computer. However, to instantiate the metagraph and represent it using a Python dictionary requires more memory than we have on available hardware. NumPy arrays have a built-in limit of 32 dimensions, thus any representation of the metagraph with NumPy arrays is limited to networks with 32 arcs.

To overcome these limitations, we consider only the western portion of the Soviet railroad network, referred henceforth as the Soviet railroad sub-network (Figure 4.13). We use this network to validate the performance of the discussed algorithms and begin by computing

the graph theoretic measures to determine potential network similarities with the example networks.

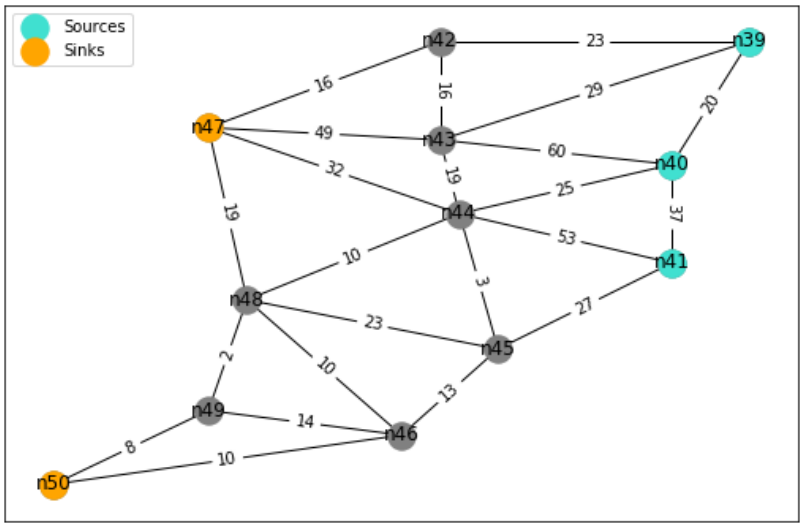


Figure 4.13. Soviet Railroad Sub-Network

The sub-network of the Soviet railroad has properties similar to that of the Simple2 network, as seen in Table 4.2 and Figure 4.14. The ABC for both Simple2 and Soviet sub-network falls completely within the $[0, 0.2]$ interval. For these two networks, the radius is approximately half of network diameter. The primary difference between the networks is the distribution of the $s - t$ subsetted ABC where the distribution for the Simple2 network ranges between 0 and 0.25, while the distribution for the Soviet sub-network is spread from 0 to 0.9. We also show the maximum flow solution, $p(x)$, for the three networks.

Table 4.2. Soviet Railroad Network - Graph Theoretic Properties.

Network $G = (N, A)$	Simple2	Soviet rail network	Soviet sub-network
$\|N\|$ or $order$	9	43	12
$\|A\|$ or $size$	14	94	23
$diam(G)$	4	8	5
$rad(G)$	2	4	3
$\lambda(G)$	2	2	2
$\kappa(G)$	2	2	2
$p(x)$	9	163	134

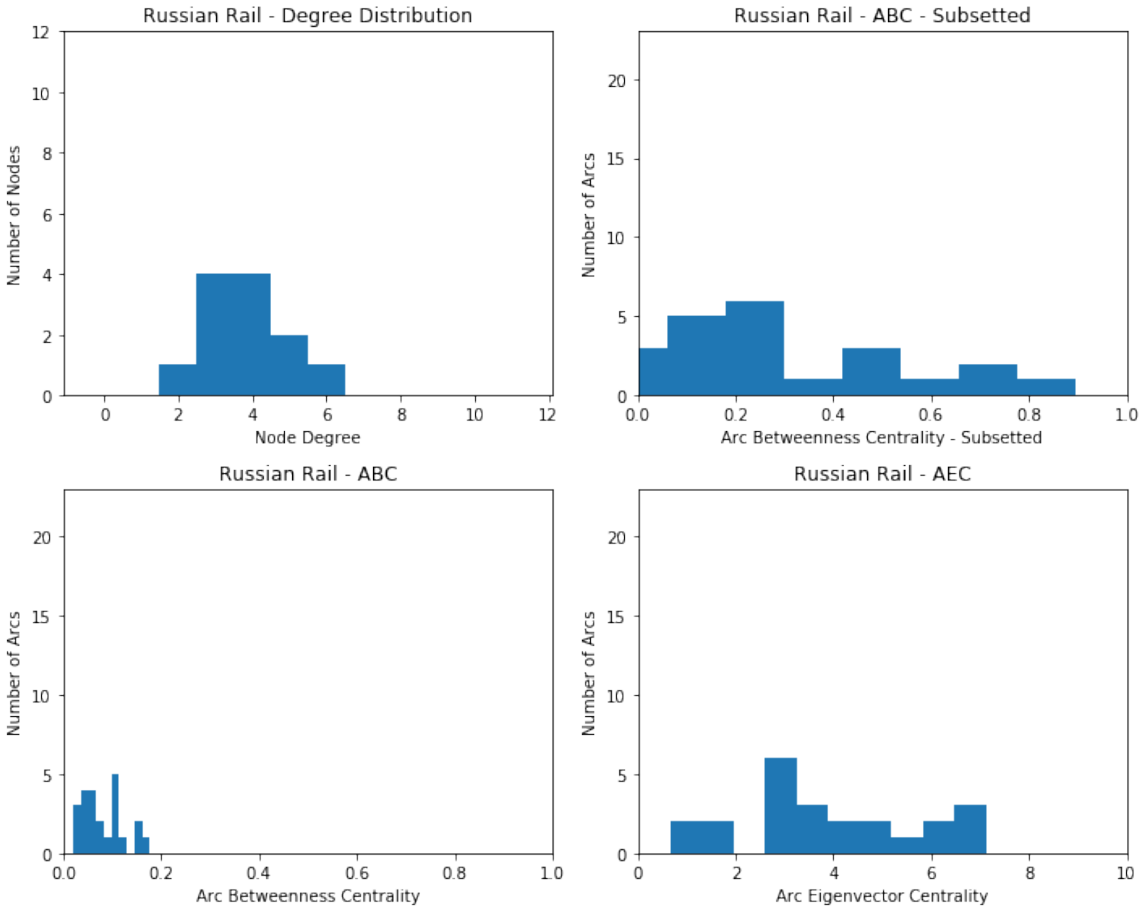


Figure 4.14. Soviet Rail Sub-Network Graph Theoretic Measures.

4.2.1 Results for BST Algorithm

As discussed previously in Section 3.2.3, we mention implementing BST in three different ways: (1) with a Python dictionary for the metagraph and a Python list for the queue; (2) with a Python dictionary for the metagraph and a Python set for the queue; (3) with an m-dimensional array for the metagraph using the NumPy library.

Here, we focus on an implementation of the metagraph as a Python dictionary and the queue as a Python list.

It takes approximately one and a half minutes to conduct BST after taking two and a half minutes to instantiate the two dictionaries that represent the green and red metagraph vertex

adjacency sets, for a total duration of approximately four minutes and 20 GB of RAM. For comparison, conducting exhaustive enumeration, using the Edmonds-Karp algorithm, takes approximately 27 minutes and about one GB of RAM on the same machine.

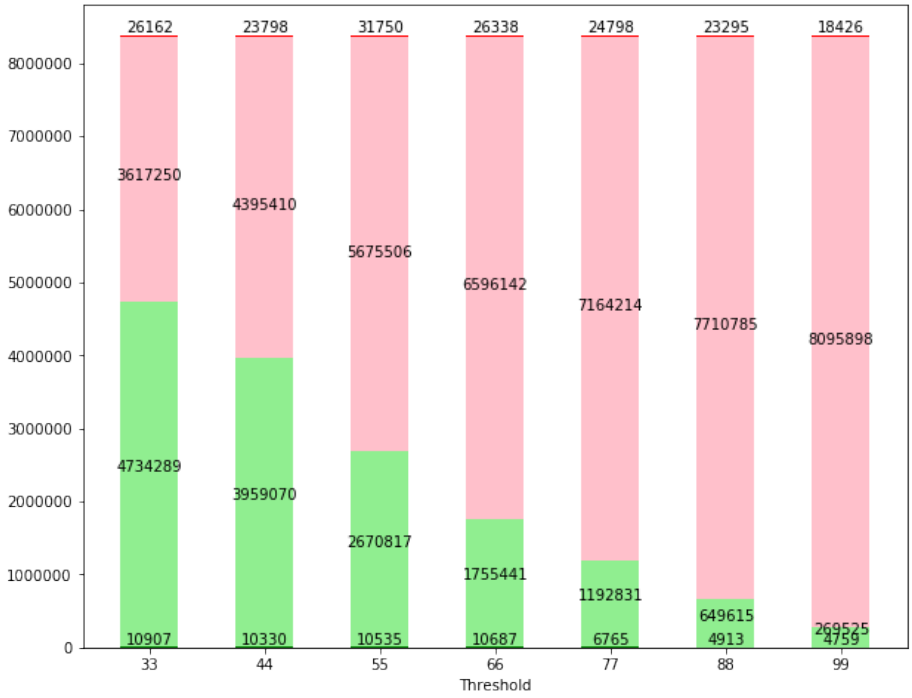


Figure 4.15. Soviet Railroad Sub-Network - BST.

In principle, our BST should perform better, given that it results in fewer performance evaluations.

In practice, the issue is whether or not our implementation of the BST requires more overhead that potentially negate the other computational savings.

Our baseline implementation uses Python dictionaries to represent the metagraph and a list to represent the queue. One alternative implementation, which uses a set for the queue, loses efficiency because it does not traverse the boundary in an orderly manner.

4.2.2 A Closer Comparison to Exhaustive Enumeration

The reader may wonder if the dictionary implementation of BST necessarily performs better than complete enumeration. This is an important question that we address. We show that for networks with at least four arcs, BST is necessarily faster than complete enumeration.

Let $m = |A|$, $n = |N|$, and x be the number of states that are not enumerated. Since Edmonds-Karp algorithm is $O(nm^2)$ for any given state, complete enumeration is $O(nm^2 \times 2^m)$, because there are 2^m states. BST has three components for computational cost: First, BST creates the adjacency dictionaries, it then uses the Edmonds-Karp algorithm to find and traverse the boundary, and finally must mark the states that are not evaluated. The computational cost for creating the adjacency dictionaries is $O(m2^m)$ since there are 2^m states and each state has m adjacent states. Therefore, BST is $O(m2^m + nm^2 \times (2^m - x) + x)$ or, equivalently, $O((nm^2 \times 2^m) - (xnm^2 - x - m2^m))$. To make our efforts worthwhile, we want $xnm^2 - x - m2^m \geq 0$, or $x \geq \frac{m2^m}{n(m^2-1)}$.

Suppose the boundary is in the middle of the metagraph, which maximizes the number of states along the boundary. Evaluating the states to reach the boundary is at most $\frac{m}{2}$, while the largest number of states along the boundary is at most $2\times\binom{m}{\frac{m}{2}}$. Therefore, $x \geq 2^m - \frac{m}{2} - 2\times\binom{m}{\frac{m}{2}}$.

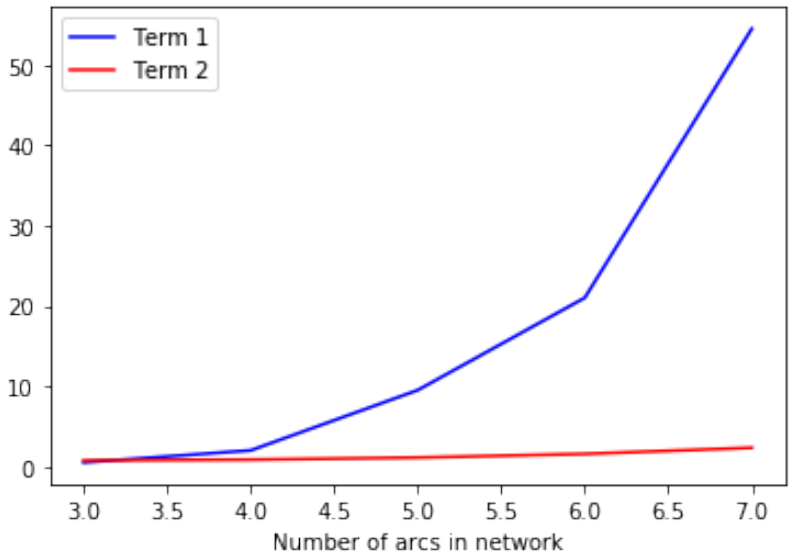


Figure 4.16. Computational Order Term Comparison of BST. Term 1 represents the savings by not enumerating states because of BST, while term 2 represents the computational cost to instantiate the metagraph adjacency dictionaries necessary for BST. Benefit from BST occurs when networks have four or more arcs.

To compare $2^m - \frac{m}{2} - 2 \times \binom{m}{\frac{m}{2}}$, referred to as term 1 in Figure 4.16, and $\frac{m2^m}{n(m^2-1)}$, referred to as term 2, we make the worst case assumption that n is minimized in order to maximize term 2, so $n = m + 1$. We can see in Figure 4.16 that term 1 is larger than term 2 for $m \geq 4$. Therefore, for $m \geq 4$, $x \geq 2^m - \frac{m}{2} - 2 \times \binom{m}{\frac{m}{2}} \geq \frac{m2^m}{n(m^2-1)}$, so $O(nm^2 \times 2^m) \geq O((nm^2 \times 2^m) - (xnm^2 - x - m2^m))$.

4.2.3 Comparison to Sub-Tree Marking

The Soviet Railroad sub-network has 23 arcs, which corresponds to $2^{23} = 8,388,608$ total vertices in the corresponding metagraph.

In Figure 4.17, we compare the amount of red and green states that need to be evaluated for the red sub-tree, green sub-tree, and BST algorithms on the Soviet Railroad sub-network network. For low performance thresholds, $\tau < 43$, the Soviet Railroad sub-network has more green states than red states.

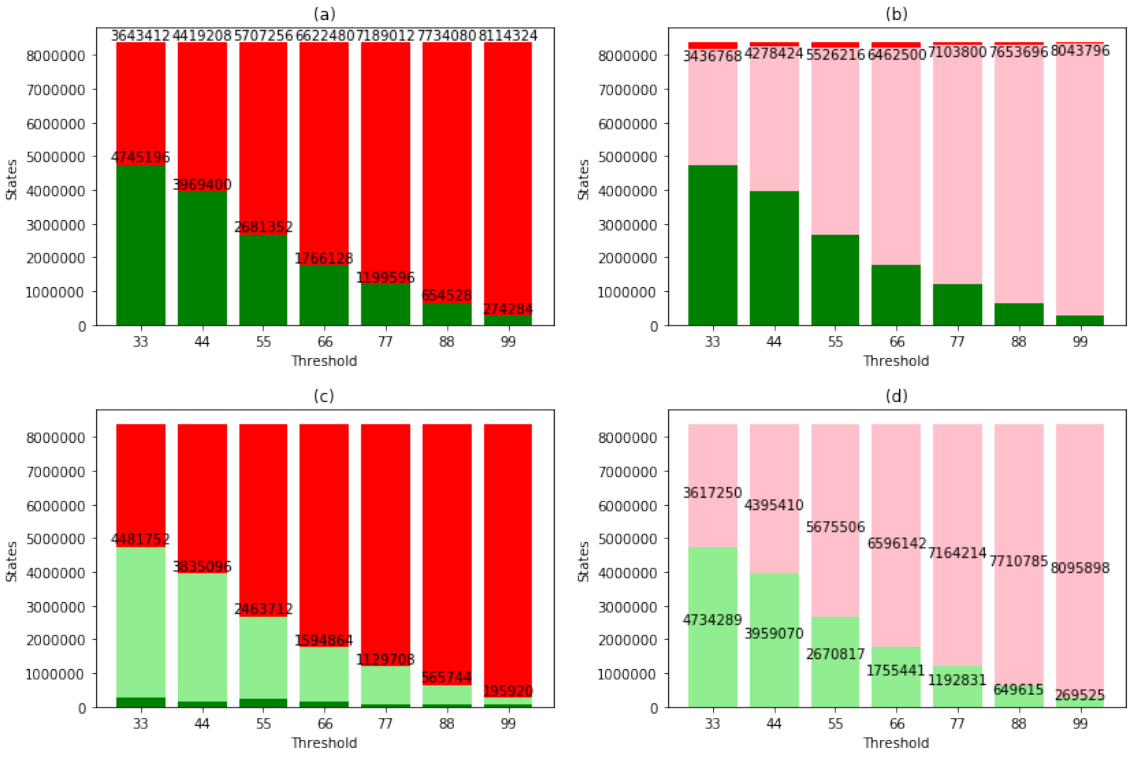


Figure 4.17. Algorithm Performance Comparison - Soviet Railroad Sub-Network. Figure 4.17a shows how many red and green states are evaluated by conducting exhaustive enumeration. Figure 4.17b highlights the savings from the red sub-tree method, ranging from 3,436,768 to 8,043,796. Figure 4.17c highlights the savings from the green sub-tree method, ranging from 195,920 to 4,481,752. Figure 4.17d highlights the savings from BST, ranging from 8,346,323 to 8,365,423.

The relative performance of the algorithms on the Soviet Railroad sub-network resembles the performance on the smaller networks where the red sub-tree method performs well for higher thresholds where the metagraph is more 'red' and the green sub-tree method performs well on lower thresholds where the metagraph is more 'green'. As in the example networks, BST requires the fewest evaluated states.

4.2.4 Comparing Metagraph Implementations

In general, we expect that the efficiency of the BST algorithm (in terms of the number of states that have to be evaluated) will be the same across different implementations of

data structures for the metagraph and queue in the algorithm itself. In practice, we observe that using an ordered list for the queue does show slight improvement over an unordered set. However, the total number of state evaluations is approximately the same whether we implement the metagraph as a Python dictionary or a NumPy n-dimensional array. However, we do see differences in the runtime performance of these implementations.

The second implementation, which uses NumPy arrays for the metagraph, is quick to mark states, but slows when about half of the states are marked, presumably because it re-marks states unnecessarily and the computational cost of filtering the arrays is more expensive than the constant-time check in the primary implementation. The overall runtime for the array implementation of BST is approximately 40 minutes and uses about one gigabyte of RAM even though it enumerates about the same amount of states as the Python dictionary implementation.

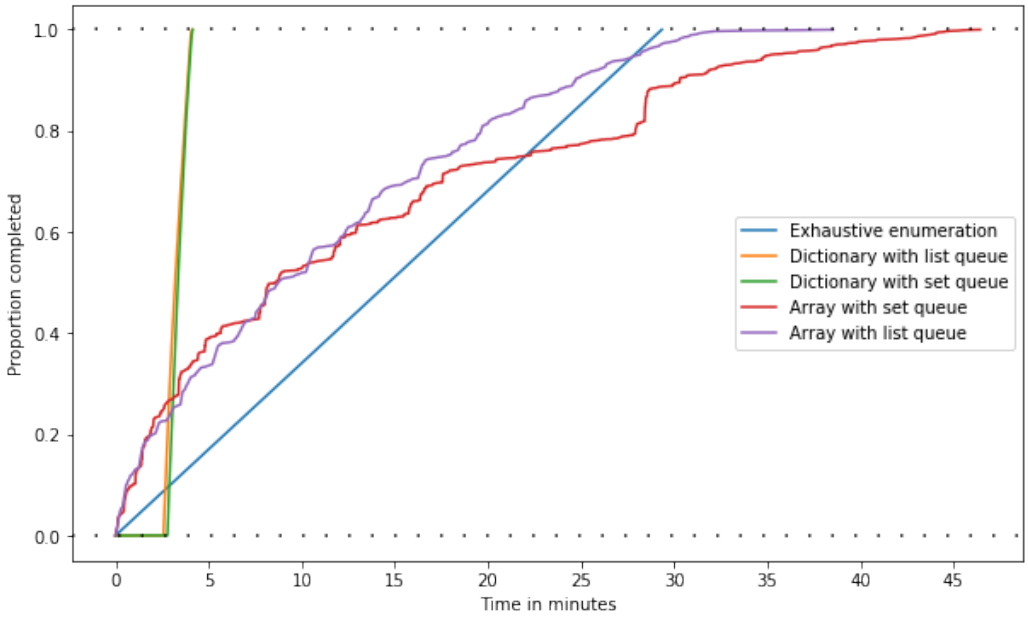


Figure 4.18. Time Comparison For Implementation Methods. Exhaustive enumeration is shown in blue and has a constant performance that is directly related to the Maximum Flow algorithm used and the size of the network. Python dictionary implementations are shown in orange and green and depict the initialization time where they begin enumeration after 2.5 minutes. The NumPy array implementations are shown in red and purple and show the degrading performance of the implementation, presumably because of the computational cost to filter the arrays begins to outweigh the benefit of excluding vertices from the enumeration algorithm.

4.3 Discussion

Exhaustive enumeration requires enumeration of all possible network states. Although this may not be particularly slow for small networks with simple optimization problems, this method becomes increasingly costly for large networks or for networks with difficult optimization problems. The "smart enumeration" methods employed by Alderson and Carlyle (2017) provide significant improvements over exhaustive enumeration and the $s-t$ subsetted ABC ordering performs as well or better than a random ordering when using the red sub-tree algorithm. This method does not require much additional memory and requires fewer enumerations than exhaustive enumeration. BST, when implemented with Python

dictionaries, is much faster than exhaustive enumeration, but have additional memory requirements. Implementing BST with a NumPy ndarray reduces these memory requirements at the expense of additional comptuational cost when filtering the 2^m arrays.

CHAPTER 5:
Conclusion and Future Work

5.1 Conclusion

Traditional network flow problems identify optimal solutions for the attacker and for the defender, but do not consider an actor beginning as the attacker and ending as the defender, a concept critical to planning military operations. Barrow (2019) introduced the concept of network shaping to address this deficiency and reveal other useful states. This thesis examines different techniques to enumerate the metagraph of states to reveal useful states that are not typically identified by traditional methods. Graph theoretic heuristic measures show promise in selecting better state space ordering for the green and red sub-tree methods implemented in Alderson and Carlyle (2017). Those methods require limited computer memory and the simple monotonic assumption that network performance does not degrade after a repair, nor does it improve after an interdiction. They are a significant improvement over exhaustive enumeration, but, when employed individually, can only prune in one direction and may unnecessarily enumerate a portion of the metagraph.

By incorporating both directions, BST quickly partitions the metagraph into green and red states for a particular threshold. The main disadvantage of BST is that it requires more memory than exhaustive enumeration and the green and red sub-tree methods because of the metagraph dictionaries. Although the additional memory requirements introduce a new problem, BST brings the concept of network shaping, introduced in Barrow (2019), closer to operational employment.

5.2 Future Work

Future work can examine improvements to the methods proposed in this thesis or alternative methods that utilize the conceptual underpinnings.

5.2.1 BST Improvements

Since metagraph structure is consistent for a given number of network arcs, future work could reduce the preliminary computation to create the green and red adjacency dictionaries by having a reference database that contains the metagraph structure. Additionally, future work should examine retaining the performance values that prune states in order to create performance intervals for each state that get updated after iterative runs of BST for different threshold values. This information could also be implemented to reduce enumerating states when the new threshold does not fall within the performance interval of the states. A third modification to BST could exclude having metagraph adjacency dictionaries, but would need to consider how to avoid re-determining state adjacencies, which increases computational time.

5.2.2 Machine Learning

Machine learning could incorporate graph theoretic heuristics to quickly approximate network operating states that are above or below particular performance thresholds.

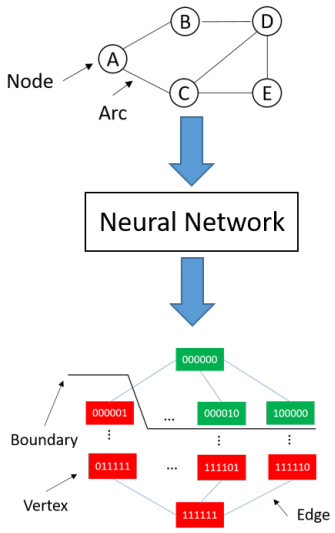


Figure 5.1. Neural Network Methodology. The goal for using a neural network is to ultimately provide a flow network and receive the output of the metagraph.

An alternate goal of using machine learning would be to reduce the computational complexity of each performance calculation by approximating solution values for the max flow

problem. This could be done directly, by approximating the max flow solution based on the characteristics of the flow network, or indirectly by approximating the solution based on the values of neighboring states in the metagraph.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

Ahuja R, Magnanti T, Orlin J (1993) *Network flows: Theory, algorithms, and applications* (Prentice Hall, New Jersey).

Alderson D, Brown G, Carlyle W (2014) Assessing and improving operational resilience of critical infrastructures and other systems. *Tutorials in Operations Research (to appear)* (Hanover, MD: Institute for Operations Research and Management Science).

Alderson D, Brown G, Carlyle W, Wood RK (2011) Solving defender-attacker-defender models for infrastructure defense. Wood K, Dell R, eds., *Operations Research, Computing and Homeland Defense*, 28–49 (Hanover, MD: Institute for Operations Research and the Management Sciences).

Alderson D, Brown G, Carlyle WM (2015) Operational models of infrastructure resilience. *Risk Analysis* 35(4):562–586.

Alderson D, Carlyle WM (2017) Enumeration and bounding arguments for infrastructure resilience analysis. Presentation, INFORMS Computing Society, Austin, TX.

Arasteh M, Alizadeh S (2019) A fast divisive community detection algorithm based on edge degree betweenness centrality. *Applied Intelligence* 49(2):689–702, https://doi.org/10.1007/s10489-018-1297-9.

Barrow HJ (2019) Network shaping. Master's thesis, Department of Operations Research, Naval Postgraduate School, Monterey, CA, https://calhoun.nps.edu/handle/10945/62705.

Bollobás B (1998) *Modern Graph Theory* (Springer), doi:10.1007/978-1-4612-0619-4.

Boyd J (2018) *A discourse on winning and losing* (Air University Press, Curtis E. LeMay Center for Doctrine Development and Education).

Brendecke JW (2016) Optimal repair and replacement policy for a system with multiple components. Master's thesis, Department of Operations Research, Naval Postgraduate School, Monterey, CA, https://calhoun.nps.edu/handle/10945/49422.

Brown G, Carlyle WM, Salmerón J, Wood K (2006) Defending critical infrastructure. *Interfaces* 36(6):530–544,616,618.

Brown GG, Dell RF (2007) Formulating integer linear programs: A rogues' gallery. *INFORMS Transactions on Education* 7(2):153–159.

Chartrand G, Zhang P (2013) *A first course in graph theory* (Courier Corporation).

Christiano P, Kelner J, Madry A, Spielman D (2010) Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. *arXiv.org* ISSN 2331-8422, URL http://search.proquest.com/docview/2087563948/.

Clark CR (2017) The threshold shortest path interdiction problem for critical infrastructure resilience analysis. Master's thesis, Department of Operations Research, Naval Postgraduate School, Monterey, CA, https://calhoun.nps.edu/handle/10945/56115.

Daitch S, Spielman D (2008) Faster approximate lossy generalized flow via interior point algorithms. *arXiv.org* ISSN 2331-8422, http://search.proquest.com/docview/2090538945/.

Diestel R (2016) *Graph Theory* (Springer).

Erdös P, Réyni A (1959) On random graphs I. 290–297.

Gharehbaghi K (2016) Artificial neural network for transportation infrastructure systems. *MATEC web of conferences* 81:05001, https://doi.org/10.1051/matecconf/20168105001.

Girvan M, Newman ME (2002) Community structure in social and biological networks. *Proceedings of the national academy of sciences* 99(12):7821–7826.

Hammer B, Jain BJ (2004) Neural methods for non-standard data. *proceedings of the 12 th European Symposium on Artificial Neural Networks (ESANN 2004), d-side pub*, 281–292.

Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del R'ıo JF, Wiebe M, Peterson P, G'erard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE (2020) Array programming with NumPy. *Nature* 585(7825):357–362, URL http://dx.doi.org/10.1038/s41586-020-2649-2.

Harris T, Ross F (1955) Fundamentals of a method for evaluating rail net capacities. Technical report, RAND CORP SANTA MONICA CA.

Hart S, Klosky J, Katalenich S, Spittka B, Wright W (2014) Infrastructure and the operational art: A handbook for understanding, visualizing, and describing infrastructure systems. Technical Report ERDC/CERL TR-14-14, US Army Engineer Development and Research Center.

Joint Chiefs of Staff (2017) Joint planning. JP 5-0, Washington, DC, https://www.jcs.mil/Portals/36/Documents/Doctrine/pubs/jp5_0_20171606.pdf.

Joint Chiefs of Staff (2018) Joint operations. JP 3-0 ch 1, Washington, DC, https://www.jcs.mil/Portals/36/Documents/Doctrine/pubs/jp3_0ch1.pdf?ver=2018-11-27-160457-910.

Knox SW (2018) *Machine learning : a concise introduction* (Wiley, New Jersey).

Latouche P, Rossi F (2015) Graphs in machine learning: an introduction. *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), Proceedings of the 23-th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2015)*, 207–218.

Mishkovski I, Kojchev R, Trajanov D, Kocarev L (2010) Vulnerability assessment of complex networks based on optimal flow measurements under intentional node and edge attacks. *ICT Innovations 2009*, 167–176, https://doi-org.libproxy.nps.edu/10.1007/978-3-642-10781-8_18.

Moon S, Lee JG, Kang M (2014) Scalable community detection from networks by computing edge betweenness on mapreduce. *2014 International Conference on Big Data and Smart Computing (BIGCOMP)*, 145–148, https://doi.org/10.1109/BIGCOMP.2014.6741425.

Newman MEJ (2018) *Networks: An Introduction,* 2nd ed. (Oxford University Press, New York).

Oltman CB, Davidson J William T, Kempf SS, Moore TK, Ogren TP (1996) Interdiction: Shaping things to come. Technical report, http://www.dtic.mil/docs/citations/ADA332935.

Rao V, Rao S (2012) Application of artificial neural networks in capacity planning of cloud based it infrastructure. *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 1–4, https://doi.org/10.1109/CCEM.2012.6354597.

Schulze C (2014) A comparison of techniques for optimal infrastructure restoration. Master's thesis, Department of Operations Research, Naval Postgraduate School, Monterey, CA, https://calhoun.nps.edu/handle/10945/44665.

Zhang W, Chien J, Yong J, Kuang R (2017) Network-based machine learning and graph theory algorithms for precision oncology. *npj Precision Onc* 1(1):25, ISSN 2397-768X, URL https://www.nature.com/articles/s41698-017-0029-7.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California